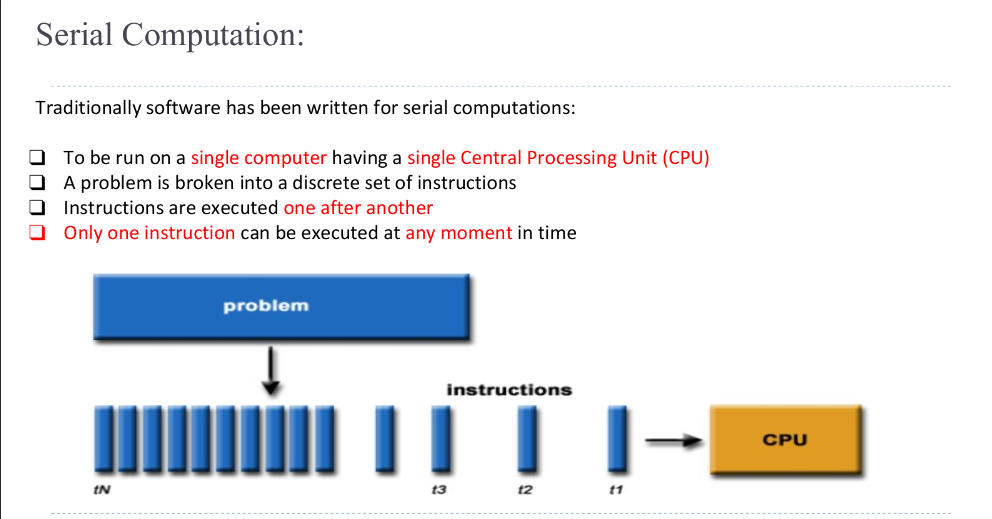
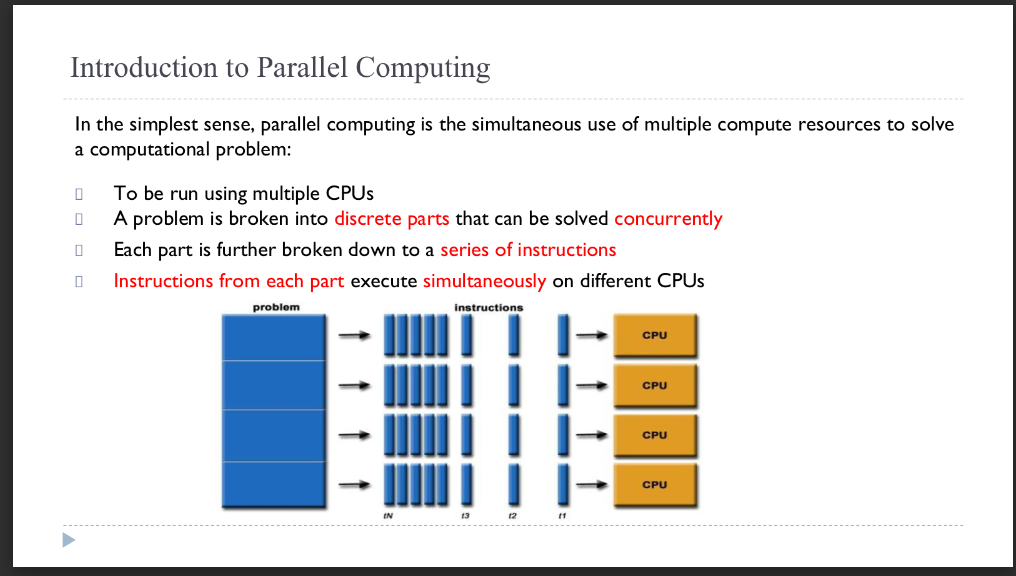
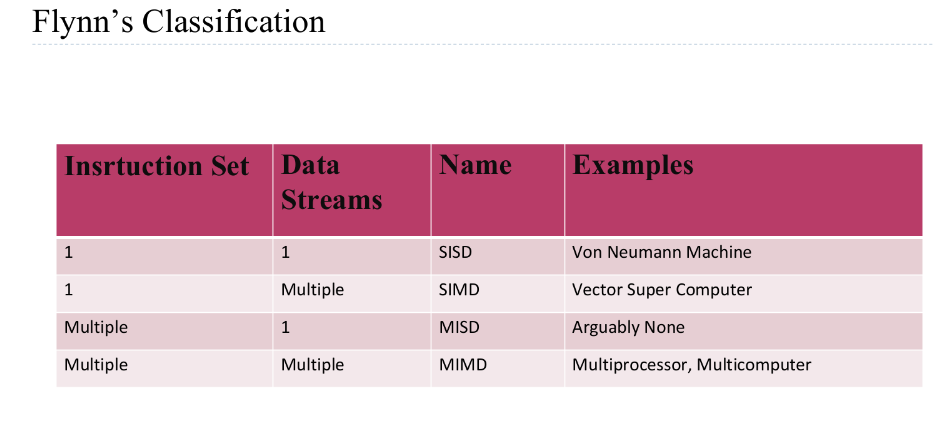
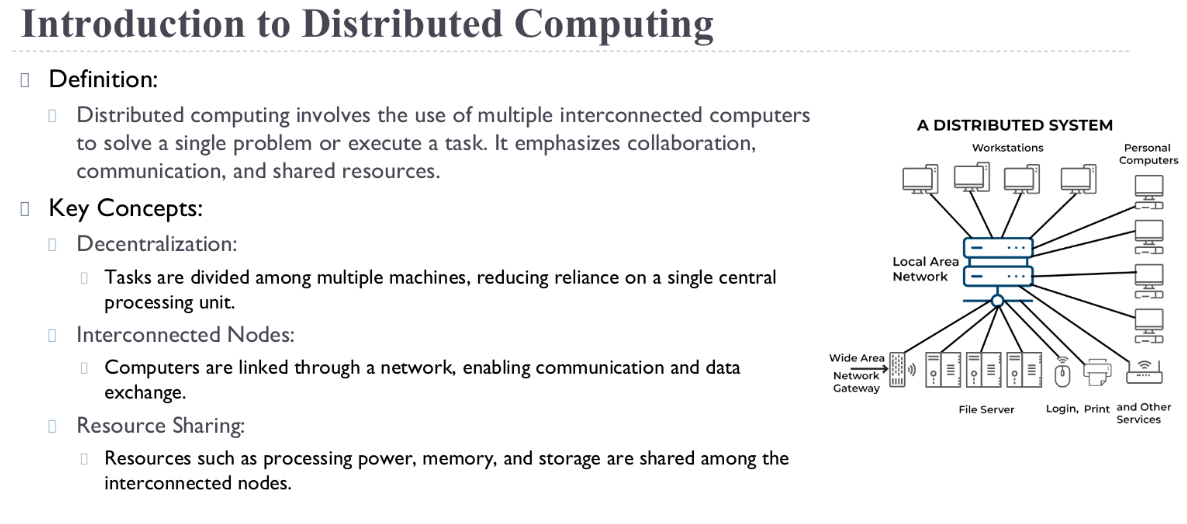
# **Lecture 1:**







**Details in hand written notes.**





## 

## List of things to consider while Designing parallel programs/ communication:

1. **Cost of Communication:**

When tasks in a parallel program need to share data, it involves some overhead.

Think of it like sending messages between people. Each message takes time to compose, send, and receive. Similarly, in a parallel program, sending data between tasks consumes computational resources and time.

Also, when tasks need to synchronize, meaning they need to wait for each other to finish their part before continuing, it can slow down overall progress. It's like waiting for someone to finish talking before you can respond.

1. **Latency vs. Bandwidth:**

**Latency** is like the delay between sending a message and receiving a response. It's the time it takes for a small piece of information to travel between tasks.

**Bandwidth**, on the other hand, is like the capacity of a communication channel. It's how much data you can send through in a given amount of time.

If you're sending lots of small messages, like saying "hello" many times instead of saying a longer sentence, the time spent waiting for each "hello" to be sent and received can add up and become the main factor slowing down communication.

1. **Synchronous vs. Asynchronous Communication:**

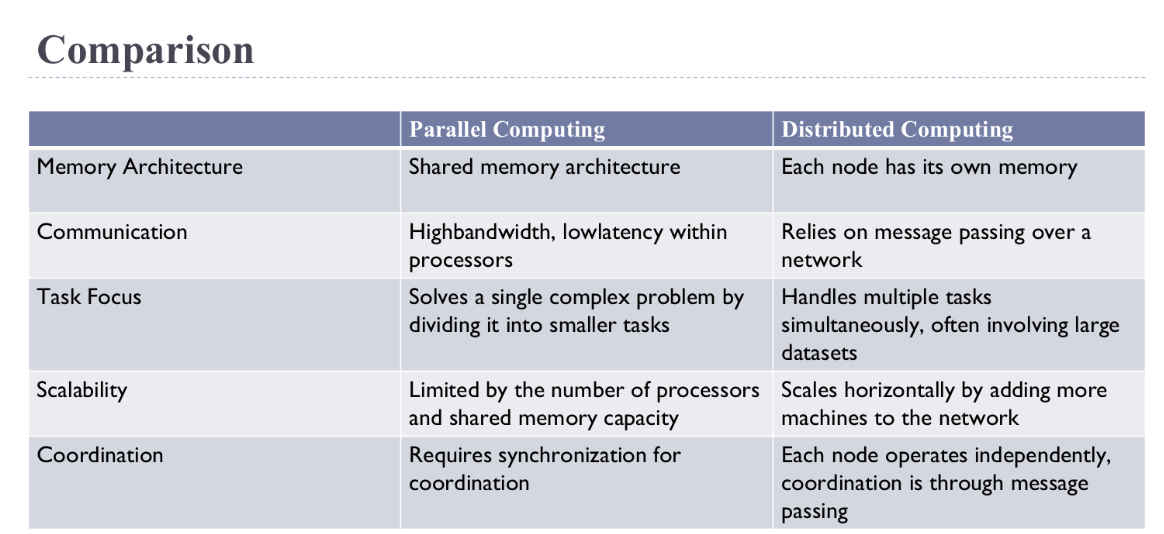
**Synchronous communication** is when tasks have to wait for each other. It's like making a phone call where you have to stay on the line until the conversation is finished. While you're talking, you can't do anything else.

**Asynchronous communication** allows tasks to continue working while waiting for a response. It's like sending a text message where you can keep doing other things while waiting for a reply. This can be more efficient because you're not sitting idle waiting for each message to be sent and received.

1. **Scope of Communication:**

**Point-to-point communication** is like having a private conversation between two specific people. Only those two parties are involved in exchanging information.

**Collective communication** involves everyone in a group. It's like having a group discussion where everyone shares their thoughts and listens to what others have to say. This type of communication is often used for tasks like synchronizing all tasks in a parallel program or gathering data from multiple sources.



## 

## **Memory Architectures:**

1. **Shared Memory:**

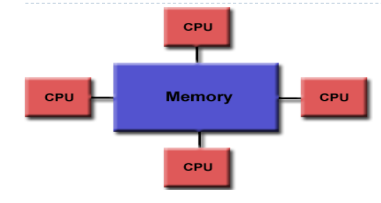
* In a shared memory system, multiple processors or CPUs have their own execution units but share the same physical memory space.
* Changes made by one CPU in the memory are immediately visible to all other processors.

**Advantages:**

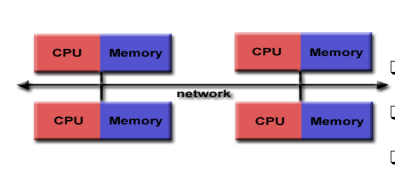
* Easy programming since all processors can directly access the same memory.
* Fast data sharing due to the proximity of memory to CPUs.

**Disadvantages:**

* Scalability issues arise when adding more CPUs, as it increases traffic on the shared memory-CPU path.
* Programmers must ensure proper synchronization and access to global memory.



1. **Distributed Memory:**

****

In a distributed memory system, each processor has its own local memory, and changes made by one processor don't affect others. Communication between processors is needed to exchange data.

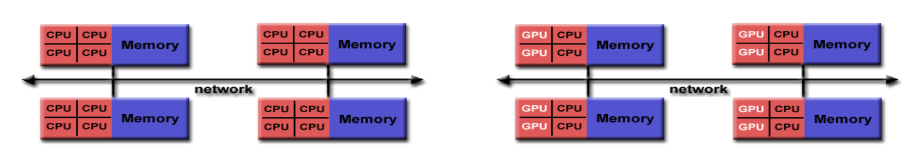
**Advantages:**

* Scalability, as memory scales with the number of CPUs.
* Each CPU can rapidly access its own memory without overhead related to global cache coherence.

**Disadvantages:**

* Programmers need to manage data communication between processors.
* Mapping existing data structures to this memory organization can be challenging.

1. **Hybrid Distributed Shared Memory:**

****

* This architecture combines elements of both shared and distributed memory systems.
* Typically found in large-scale supercomputers where both shared memory machines and distributed memory machines are used.
* Each compute node has shared memory among its processors, but communication is needed between compute nodes to exchange data.

**Advantages and Disadvantages:**

* Inherits advantages and disadvantages from both shared and distributed memory architectures.
* Offers increased scalability but introduces increased programming complexity due to the need to manage both shared and distributed memory aspects.

# 

# **Lecture 2:**

## **Why is Time Important?**

Time is super important for a couple of big reasons:

**E-commerce Transactions:** Think about all those transactions happening online - buying stuff, transferring money, etc. Computers need to know exactly when these things happen, so they all agree on the sequence of events. Like, you buy a book online, and both your computer and the seller's computer need to agree on the time you clicked "buy"!

**Understanding Distributed Systems**: When you have computers all over the place talking to each other (like in big networks or the internet), understanding time helps us figure out the order of things. Like, did this message get sent before or after that message? Time helps us make sense of it all.

## **Problem with Physical Time in Distributed Systems**

Imagine each computer has its own clock, ticking away. But here's the tricky part: these clocks don't all tick at exactly the same speed, and they don't start at the same time. So, even if we try to synchronize them, they'll always be a bit off.

**What We'll Cover**

**Synchronizing Clocks:** We'll talk about ways to get these clocks to agree on the time as closely as possible, even if they can't be perfect.

**Logical Clocks:** These are like special tools for figuring out the order of events in a distributed system. We'll learn about something called **vector clocks**, which help us keep track of what happened when.

## **Physical Clocks in Computers**

Inside a computer, there's a special gadget (clock) that counts something called **oscillations**. Imagine it like a super fast heartbeat. This gadget uses a crystal and counts how many times it beats in a second. It keeps track of this count using a register, which is like a storage place.

These clocks can also be set up to interrupt the computer at regular times, like a reminder to switch tasks. This helps with things like **timeslicing**, where different programs take turns using the computer.

**Making Sense of the Clock**

When the computer's operating system wants to know the time, it reads the count from the clock gadget. Let's call this count **Hi(t)**, where "i" is the computer's number. But, this count isn't exactly what we humans think of as time. So, we do some math to scale it and add or subtract a bit to get something closer to real time. This gives us **Ci(t)**, our **software clock**, which we use to measure time on that computer.

For example, Ci(t) might be a really big number representing how many nanoseconds have passed since a certain starting point.

But here's the catch: these clocks aren't perfect. They can be a bit off because of things like:

## **Why Clocks Aren't Perfect**

Even though we try to make our clocks as accurate as possible, they're not perfect. Sometimes, two clocks won't show the exact same time, which we call **clock skew**. Also, these clocks can drift **(clock drift)**, which means they start counting time at slightly different speeds. This happens because the crystals they're based on can vary in how they're made or even with temperature changes.

So, even though we try to keep our clocks in sync, there'll always be some differences because of how they're made and used.

**Clock Skew**: This is when two clocks aren't showing the exact same time at the same moment.

**Clock Drift:** Sometimes, these clocks just don't keep time at the same rate, so they start showing different times.

## **Why Clocks Drift**

So, remembe r those little oscillations we talked about earlier? Well, even though they're super tiny, if you let them keep going for a long time, they add up. Imagine you're counting each heartbeat, but your count gets a tiny bit off each time. After a while, you'll notice a difference between your count and someone else's count.

**Drift Rate**

The drift rate is like how fast this difference between clocks adds up over time. For regular quartz clocks, this drift rate is about 1 millionth of a second every second. That means, if you leave two clocks running for about 11.6 days, they'll be off by a whole second!

## **Synchronising Physical Clocks:**

**External Synchronization**

This is when we sync up our clocks with an outside source that we trust a lot. It's like when you set your watch by looking at the time on your phone, which you trust to be accurate. We do this because we want all our clocks to agree on what time it is. This helps when we need to know exactly when things happened, like for keeping track of transactions or events in our distributed system.

In this mode, we're making sure that each of our clocks (let's call them Ci) is really close to the time given by a super trustworthy source called UTC. We set a limit, let's say D seconds, and we want all our clocks to be within D seconds of UTC time. So, no matter when you check, all our clocks should be pretty accurate according to this time source.

* This is about making sure all clocks (Ci) agree with an external source of time (S), like UTC.
* Imagine you have several clocks, and you want them to be accurate to within a certain margin (D) of this external source.
* So, for every moment in time (t) within a certain interval (I), the difference between what the external source says (S(t)) and what each clock says (Ci(t)) should be less than D.
* In simple terms, it means all the clocks are accurate to within the margin set by D.



**Internal Synchronization**

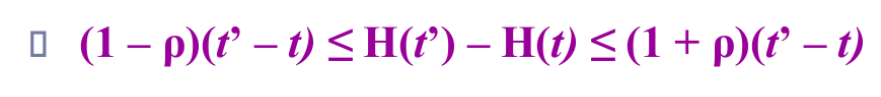
Here, instead of relying on an outside source, we make sure all our clocks inside the system agree with each other pretty well. It's like making sure all the watches in your family show almost the same time, even if they're not exactly right. This way, if something happens and we need to know the time difference between events on different computers, we can use their local clocks without worrying too much about them being way off.

* Here, instead of comparing our clocks to an outside source, we're comparing them to each other. We still have that limit D seconds, and what we want is for all our clocks to be within D seconds of each other. So, even if they're not perfectly in sync with the outside world, they're all ticking together nicely.
* Here, we're focusing on making sure all clocks within the system agree with each other pretty closely.
* So, for every moment in time (t) within a certain interval (I), the difference between any two clocks (Ci(t) and Cj(t)) should be less than a margin (D).
* It's like making sure all the clocks in your house show roughly the same time, even if they're not perfectly right.



## **Correctness of Physical Clocks:**

* A hardware clock is considered correct if its rate of drift (how much it gains or loses time) is within a certain known limit (ρ).
* This means that if you measure the time between two real moments (t and t'), the difference in what the clock says (H(t') - H(t)) should be within a range determined by the drift rate.



## **Monotonicity:**

* This means a clock always moves forward in time. It never goes backward.
* It's like your watch always ticking forward. If it ran fast, you can't turn it back to fix it.

## **Synchronisation in a synchronous system:**

Imagine you're working in a synchronized factory where every task needs to happen at specific times. Each worker (task) has their own stopwatch (clock) to keep track of time.

Now, let's say Worker A needs to pass a message to Worker B, and it's crucial that Worker B knows exactly when the message was sent.

1. **Knowing the Drift Rate:**

In this factory, everyone knows that their stopwatches can gain or lose a bit of time over a period. Let's say they've measured this drift rate and found it's about 1 second per hour.

1. **Sending the Message:**

Worker A sends a message to Worker B, including the time according to their stopwatch (let's say it's 12:00:00).

1. **Setting the Receiving Clock:**

Ideally, Worker B could just set their stopwatch to 12:00:00 plus the time it takes for the message to travel (let's say it's 1 minute).

So, Worker B would set their stopwatch to 12:01:00. Simple, right?

1. **Dealing with Variations:**

* However, in reality, message transmission times can vary. Sometimes it might take less than a minute, sometimes more.
* To account for this variability, Worker B takes a cautious approach. Instead of setting their stopwatch to exactly 1 minute after the message was sent, they set it to the halfway point between the minimum and maximum possible transmission times.
* Let's say the minimum transmission time they've ever measured is 50 seconds, and the maximum is 70 seconds. So, the halfway point is (50 + 70) / 2 = 60 seconds.
* Worker B sets their stopwatch to 12:00:30, which is halfway between 12:00:00 (when the message was sent) and 12:01:10 (the maximum possible arrival time of the message).
* By setting their stopwatch to the halfway point between the minimum and maximum transmission times, Worker B ensures that even if the actual transmission time varies, their stopwatch won't be too far off from the actual time the message was sent. This helps maintain synchronization in a system where precise timing is crucial.

**Example:**

* Worker A sends the message at 12:00:00.
* The minimum transmission time recorded is 50 seconds, and the maximum transmission time recorded is 70 seconds.
* To find the halfway point between these times:
* Add the minimum and maximum transmission times: 50 seconds + 70 seconds = 120 seconds.
* Divide the total time by 2 to find the halfway point: 120 seconds / 2 = 60 seconds.
* Add this halfway time (60 seconds) to the time when Worker A sent the message (12:00:00):
* 12:00:00 (hour doesn't change) + 1 minute (or 60 seconds).
* Therefore, Worker B sets their stopwatch to 12:00:00 + 1 minute = 12:01:00.
* So, Worker B sets their stopwatch to 12:01:00, which is halfway between the earliest possible arrival time (12:00:50) and the latest possible arrival time (12:01:10), ensuring that their stopwatch is synchronized as accurately as possible with the time when Worker A sent the message.

## **Cristian’s method for synchronizing clocks:**

**Using a Time Server**

Cristian came up with a clever way to sync up computer clocks using a time server. This server is connected to something super accurate, like a UTC source, which gives the exact time.

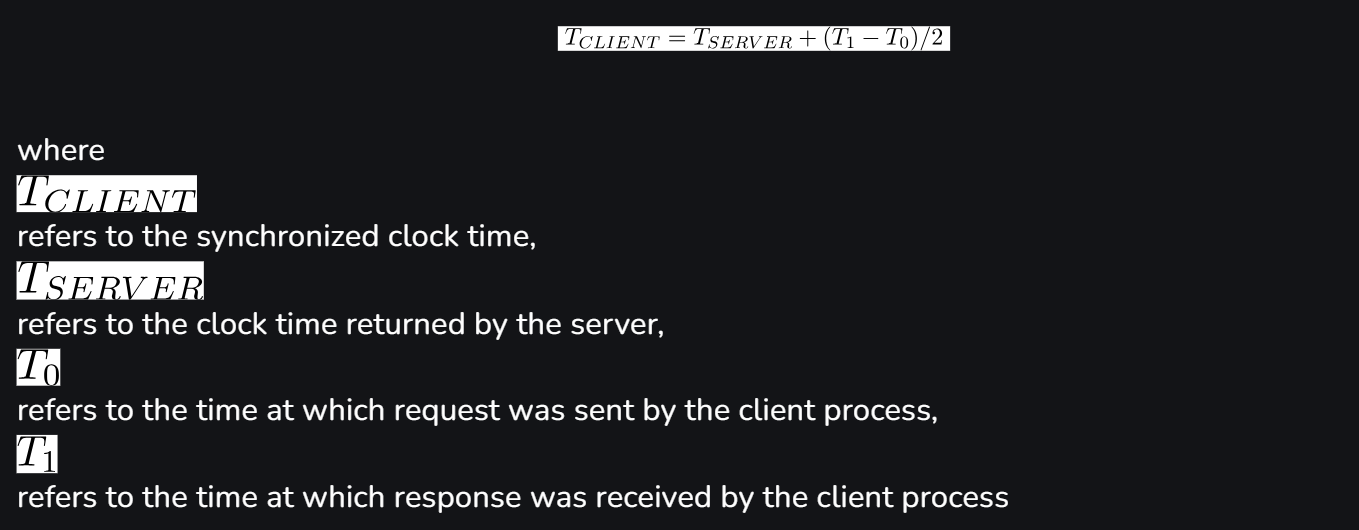
**Here's how it works:**

**Requesting Time:** When a computer (let's call it process p) wants to know the time, it asks the server (process S).

**Server Response:** The server responds with the current time, but it also includes the time it sent the response.

**Calculating Round-Trip Time:** Process p measures how long it took for the request to go to the server and for the response to come back. This total time is called the round-trip time (Tround).

**Setting the Clock:** Process p adjusts its clock based on this round-trip time. It sets its clock to halfway between when it sent the request and when it got the response.



**Accuracy Check**

But here's the thing: for this to work well, the round-trip time needs to be short enough compared to how accurate we want the clocks to be.

For example, on a local network where things move quickly, the round-trip time might be just 1-10 milliseconds. If a clock drifts by 1 millionth of a second per second, it only changes by about 0.00001 milliseconds during this time, which is tiny and acceptable.

**Making Adjustments**

If we know the minimum transmission time, we can adjust our accuracy estimate. We figure out the earliest and latest times the server could have sent the response, then use this range to adjust our clock setting.

So, by carefully timing how long it takes to talk to the server, we can keep our clocks in sync pretty well!

**Berkeley algorithm:**

The Berkeley algorithm, developed by **Gusella and Zatti in 1989**, is a method for synchronizing the clocks of computers in a network, particularly those running Berkeley UNIX.

Here's how it works:

1. **Coordinator Selection:**

A coordinator computer is chosen to act as the master. This coordinator periodically polls the other computers in the network, known as slaves, whose clocks need to be synchronized.

1. **Clock Value Exchange:**

The slaves send their current clock values back to the coordinator.

1. **Clock Time Estimation:**

The master estimates the local clock times of the slaves by observing the round-trip times, similar to Cristian's protocol.

It averages the values obtained, including its own clock reading.

1. **Selective Averaging:**

The algorithm selects a subset of clocks that do not differ from one another by more than a specified amount.

The average is then taken from only these clocks.

1. **Synchronization:**

By averaging the clock readings from multiple sources, the algorithm aims to mitigate individual clock tendencies to run fast or slow, achieving synchronization across the network.

1. **Experimental Results:**

Gusella and Zatti conducted experiments involving 15 computers, synchronizing their clocks to within about 20–25 milliseconds using their protocol.

They measured the local clocks' drift rates to be less than 2 x 10–5, with a maximum round-trip time of 10 milliseconds.

1. **Failover Mechanism:**

If the master fails, another computer can be elected to take over and function as the new master, continuing the synchronization process.

**Example:**

Let's illustrate the Berkeley algorithm with a simple example involving three computers: A (the coordinator/master) and B, C (slaves).

1. **Coordinator Selection:**

Computer A is chosen as the coordinator/master.

1. **Clock Value Exchange:**

Periodically, computer A sends a message to computers B and C, asking them to report their current clock values.

1. **Clock Time Estimation:**

* Computers B and C receive the message and send back their current clock values to computer A.
* Computer A records the time when it sent the message and the time it received the responses from B and C. It calculates the round-trip times for each.

1. **Selective Averaging:**

* Computer A evaluates the round-trip times and checks if the clocks of B and C differ from each other by more than a specified amount.
* Let's say B's clock reads 12:00:00 and C's clock reads 12:00:05. The difference is within the allowed tolerance.
* Computer A averages the clock readings of B and C, along with its own clock reading.

1. **Synchronization:**

* The averaged value is considered as the synchronized time for all computers in the network.
* Computer A then adjusts its clock to match the synchronized time, and it broadcasts this adjusted time to computers B and C.

1. **Example Scenario:**

* Let's assume that Computer A's clock reads 12:00:00, Computer B's clock reads 12:00:01, and Computer C's clock reads 12:00:02.
* After the synchronization process, the averaged time might be calculated as 12:00:01. This time is then set as the synchronized time for all computers.

1. **Failover Mechanism:**

* If Computer A (the master) fails, another computer can be elected to take over its role. This new coordinator will then continue the synchronization process, ensuring uninterrupted clock synchronization in the network.

**Note: NTP (Network Time Protocol) also included in this lecture.**

# **Lecture 3:**

## **Parallelism:**

Parallelism in microprocessors involves executing multiple tasks simultaneously to improve processing speed and efficiency. There are several types of parallelism:

* **Instruction-Level Parallelism (ILP)**: Involves executing multiple instructions at the same time within a single thread or process.
* **Data-Level Parallelism (DLP)**: Focuses on processing multiple data elements simultaneously using techniques like SIMD (Single Instruction, Multiple Data) or vector processing.
* **Task-Level Parallelism (TLP):** Involves executing multiple independent tasks or processes concurrently, often across multiple cores or processors.

The benefits of parallelism include:

* **Increased Throughput**: Parallel execution of tasks allows for higher throughput, meaning more tasks can be completed in a given time frame.
* **Improved Performance:** Parallelism can lead to faster execution of tasks, resulting in improved overall performance of the system.
* **Enhanced Efficiency:** By utilizing resources more effectively, parallelism can improve the efficiency of microprocessor systems.

However, parallelism also poses challenges and considerations:

* **Synchronization:** Ensuring that multiple parallel tasks coordinate and synchronize their execution properly to avoid conflicts and maintain data consistency.
* **Dependency Management:** Managing dependencies between tasks to ensure that tasks are executed in the correct order and that results are accurate.
* **Scalability:** Designing systems that can effectively scale with increasing levels of parallelism, without encountering diminishing returns or performance bottlenecks.

## **Architectural classification schemes:**

Architectural classification schemes help categorize computer architectures based on various design principles and features. Here are some common schemes:

**Instruction Set Architecture (ISA):**

* Classifies architectures based on the type and complexity of instructions the processor can execute.
* Examples include Complex Instruction Set Computing (CISC) and Reduced Instruction Set Computing (RISC).

**Memory Hierarchy:**

* Classifies architectures based on the organization and hierarchy of memory components.
* Examples include Von Neumann architecture, Harvard architecture, and Cache Memory Hierarchy.

**Pipelined Architecture:**

* Classifies architectures based on the use of pipelines for instruction execution.
* Examples include Instruction Pipelining and Superscalar Architecture.

**Parallelism:**

* Classifies architectures based on the degree of parallel processing employed.
* Examples include Single Instruction, Multiple Data (SIMD) and Multiple Instruction, Multiple Data (MIMD) architectures.

## **Principles of Pipelining:**

1. **Dividing Tasks into Stages:**

Principle: Pipelining involves breaking down the execution of instructions into discrete stages. Each stage represents a specific task in the instruction execution process.

1. **Parallel Processing of Instructions:**

Principle: Different stages of the pipeline operate in parallel, allowing multiple instructions to be in various stages of execution simultaneously. This increases throughput and overall processing speed.

1. **Continuous Flow of Instructions:**

Principle: Instructions move through the pipeline continuously. As one instruction completes a stage, the next instruction enters the pipeline, ensuring a steady flow of operations.

1. **Overlap of Execution:**

Principle: Pipelining aims to overlap the execution of multiple instructions. While one instruction is in the execution stage, another can be in the decoding stage, maximizing processor utilization.

1. **Stall and Hazard Handling:**

Principle: Pipelining may face hazards such as data dependencies or branch instructions. Techniques like instruction forwarding and branch prediction are employed to handle these hazards and prevent pipeline Stalls.

1. **Optimizing Resource Utilization:**

Principle: Pipelining optimizes the use of processor resources by allowing different stages to work concurrently. This reduces idle time and improves overall efficiency.

## **Principles of Vector Processing:**

1. **Simultaneous Processing of Data Elements:**

Principle: Vector processing involves the simultaneous execution of the same operation on multiple data elements. This is achieved through specialized vector instructions.

1. **Vector Registers:**

Principle: Vector processors use vector registers to store and manipulate multiple data elements. These registers allow efficient access to and processing of contiguous data.

1. **Vectorization of Code:**

Principle: Vector processing requires code to be written or compiled in a way that exploits the capabilities of vector instructions. Loops and operations are structured to take advantage of parallelism.

1. **Parallelism with a Single Instruction:**

Principle: Vector processors achieve parallelism by executing a single instruction on multiple data elements concurrently. This contrasts with scalar processors that operate on individual data items.

1. **Enhanced Throughput for Regular Data:**

Principle: Vector processing is particularly effective for regular and repetitive data structures, where the same operation is performed on a large set of data elements.

1. **Reduced Instruction Overhead:**

Principle: Vector processing minimizes instruction overhead by expressing operations on entire vectors with a single instruction, reducing the need for individual instructions for each data element.

1. **Efficient Memory Access:**

Principle: Vector processors often implement techniques like vector prefetching and caching to optimize memory access patterns, ensuring efficient retrieval of vector data from memory.

## **Array processors**

Array processors are specialized computers made for handling arrays or matrices of data really efficiently. Here are the main things to know about them:

## **Characteristics of Array Processors:**

1. **Parallel Processing:** They're good at doing many things at once. Each part of the processor works on a different piece of the array at the same time.
2. **Specialized Instructions:** These processors come with special instructions just for working with arrays. These instructions help them do array operations really fast.
3. **Vector and Matrix Operations:** They're great at doing math on arrays and matrices, like adding, multiplying, or transforming them all at once.
4. **Memory Architecture:** The way they store and access data is designed to be super quick. They might have special memory setups to make sure they can get the data they need fast.
5. **High Throughput:** They're really good at handling big sets of data quickly, especially when the data follows a pattern. This makes them perfect for tasks like scientific simulations or big engineering projects.
6. **Scientific and Engineering Applications:** They're used a lot in science and engineering because those fields often deal with huge amounts of data that need to be crunched quickly. So, anything from simulating physical systems to processing signals can benefit from array processors.
7. **Data Parallelism:** Array processors are built around the idea of doing the same thing to lots of data at once. This fits perfectly with how arrays work, where you often want to apply the same operation to many elements simultaneously.

Array processors have their strengths and challenges. Let's break them down:

## **Advantages:**

1. **Efficiency in Parallel Operations:** They're really good at doing lots of things at once with arrays, which makes them super fast for computations.
2. **Optimized for Mathematical Operations:** These processors are designed specifically for doing math, which is great for scientific and engineering tasks that involve lots of calculations.
3. **High Throughput:** Because they can work on many parts of the data simultaneously, they can handle big datasets really quickly.

## **Challenges:**

1. **Limited Applicability:** They're specialized machines, so they might not be the best choice for every kind of task. General-purpose processors might be better for some jobs.
2. **Programming Complexity:** Writing programs for array processors can be tricky because you have to explicitly tell the computer how to split up the work. This can make the programming more complex compared to regular programming.

## **RISC (Reduced Instruction Set Computing) Architecture:**

**What is RISC?**

* RISC stands for Reduced Instruction Set Computing.
* It uses a small, efficient set of simple instructions.
* Each instruction is designed to be executed quickly, usually in one clock cycle.

**Principles of RISC:**

* **Simplicity**: The instruction set is small and easy to understand.
* **Efficiency**: Instructions are designed to be executed very quickly.
* **Predictability**: Simple instructions make the execution process straightforward.

**Advantages of RISC:**

* **Simple Design**: Easier to design and build processors.
* **Fast Execution**: Instructions execute quickly, often in a single cycle.
* **Power Efficiency**: Uses less power, making it ideal for battery-powered devices.
* **Compiler-Friendly**: Easier for compilers (software that translates code) to optimize performance.

**Disadvantages of RISC:**

* **Increased Memory Usage:** RISC programs often need more memory due to the larger number of instructions.
* **Higher Software Complexity:** RISC architectures require more complex software for operations that CISC handles with single instructions.

**Real-World Applications:**

* **Mobile Devices**: ARM processors in smartphones and tablets.
* **Embedded Systems:** Used in small gadgets and appliances.
* **Networking Equipment:** Routers and switches for fast data processing.

## **CISC (Complex Instruction Set Computing) Architecture:**

**What is CISC?**

* CISC stands for Complex Instruction Set Computing.
* It uses a large set of complex instructions.
* Each instruction can perform multiple operations.

**Principles of CISC:**

* Complex Instructions: Designed to do more in a single instruction.
* Efficiency for Complex Tasks: Reduces the total number of instructions needed for complex operations.

**Advantages of CISC:**

* Versatility: Can perform complex tasks with fewer instructions.
* Efficiency in Certain Tasks: Better for tasks that require many operations.

**Disadvantages of CISC:**

* Complex Design: More difficult to design and build processors.
* Higher Power Consumption: More complex instructions can use more power.

**Examples of CISC Processors:**

* **x86 Processors**: Used in most desktop and laptop computers, made by companies like Intel and AMD.

# 

# **Lecture 4:**

## **Causal precedence relation:**

The causal ordering of events is based on two simple and intuitively

obvious points:-

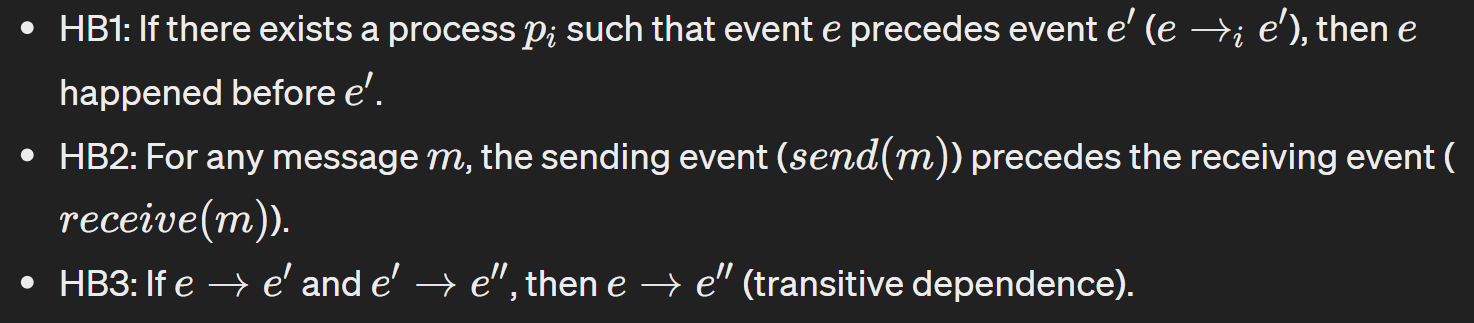
* If two events occurred at the same process pi (i = 1, 2, …, N), then they occurred in the order in which pi observes them i.e. the order →i.
* Whenever a message is sent between processes, the event of sending the message occurs before the event of receiving the message.

Lamport called the partial ordering obtained by generalizing these

two relationships the **happened-before relation.**

Also sometimes known as the relation of **causal ordering or potential causal ordering.**

The **happened-before relation**, denoted by →, defined as follows:



## **Models of communication networks**

In communication networks, different models govern how messages are delivered between processes. Here are three key models:

**FIFO Model:**

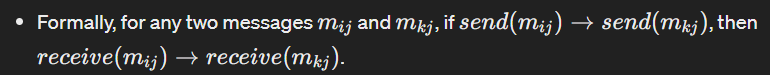
In the FIFO (First-In-First-Out) model, each channel behaves like a FIFO message queue. This means that messages sent first are received first, preserving the order in which they were sent.

**Non-FIFO Model:**

In the **Non-FIFO** model, the sender adds messages to the channel, and the receiver removes messages from the channel in a random order. This can result in messages being received out of order compared to when they were sent.

**Causal Ordering Model:**

The Causal Ordering model is based on Lamport's happens-before relation. It ensures that if one message was sent before another, then the corresponding receive events will also occur in the same order.



​

The relationship between these models is hierarchical:

* Causal Ordering (CO) is a subset of FIFO meaning that if messages are causally ordered, they will also be delivered in FIFO order.
* FIFO is a subset of Non-FIFO, indicating that FIFO order is a stricter requirement than simple message delivery without any particular order.

## **What's the Global State?**

* Imagine you have several computers (let's call them Process 1, Process 2, etc.) connected to each other, and they can send messages to communicate.
* The global state is like taking a snapshot of all these computers and their messages at a particular moment. It shows what each computer is doing and what messages are traveling between them.

**States of Processes and Channels:**

* Each computer (or process) has its own state, like what tasks it's working on or what data it's storing.
* Also, the channels between computers (where messages travel) have a state, which is basically the messages waiting to be delivered.

**Changes in State:**

* Whenever something happens, like a computer finishing a task or sending a message, it changes its state. And when a message moves from one computer to another, it also changes the state of the channel.
* So, the global state keeps track of all these changes happening across all computers and channels.

**Recording Global State:**

* Since all these computers are working independently and may not have the same time, it's tricky to capture their states at the exact same moment.
* But we want to record a consistent snapshot, meaning it should accurately represent what's happening without any confusion.
* For example, if Process 1 sends a message to Process 2, we should make sure that in our snapshot, we see the message being sent before it's received.
* Design of efficient methods for recording the global state of a distributed system is an important problem.

## **From the global state it is possible to:**

* Detect deadlock between a set of process
* Determine that an object has become garbage
* Detect that a distributed algorithm has terminated
* Debug a distributed program

For a global snapshot to be meaningful, it should be consistent:

**Consistent Global States:**

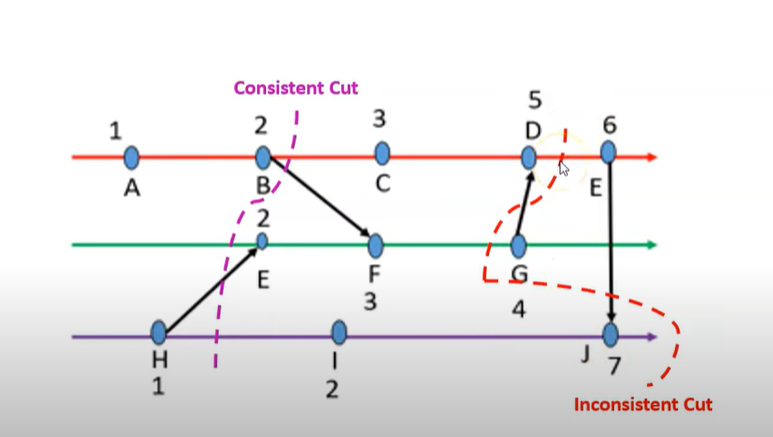
* A global state is consistent if it follows a simple rule: every message that's received should have been sent earlier. This makes sense because a message can't just appear out of nowhere; it must have been sent by someone.
* In other words, there should be a clear cause-effect relationship between sending and receiving messages.
* **For example**, if you receive a letter, there must have been someone who sent it to you.

**Inconsistent Global States:**

* An inconsistent global state breaks this rule. It's like receiving a letter without anyone having sent it. Something doesn't add up, and it's not a reliable snapshot of the system.
* **For instance**, if you find a letter in your mailbox but there's no record of anyone sending it, it's confusing and inconsistent.

## **Cuts of Distributed Computation:**

* Think of a cut as slicing through time and space, separating events that happened before the cut from those that come after.
* Each cut corresponds to a specific moment in our snapshot.
* We want our cut to be consistent, meaning it reflects the correct order of events and messages.
* Event before on left side of the cut, and after on right side.



## **Issues in recording a consistent global state**

**I1: How to distinguish between the messages to be recorded**

**in the snapshot from those not to be recorded.**

The answer comes from conditions C1 and C2 as follows:-

When we take a snapshot of the global state, we need to decide which messages should be included in that snapshot.

* **Condition C1:** Any message sent by a process before it takes its snapshot should be included in the snapshot. This ensures that all relevant messages that led to the current state are captured.
* **Condition C2**: Any message sent after a process takes its snapshot should not be included. This prevents messages that arrive after the snapshot from affecting the state, maintaining the integrity of the snapshot.

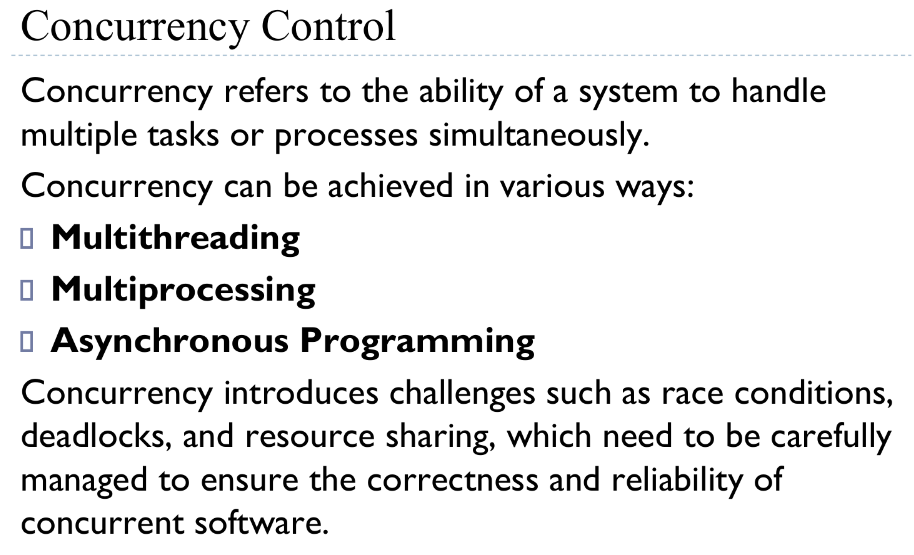
**I2: How to determine the instant when a process takes its snapshot.**

Another challenge is deciding the exact moment when a process should take its snapshot.

**Condition C2 (Continued):** To ensure consistency, a process must take its snapshot before it processes any message sent by another process after it took its snapshot.

**Note: Logical and Vector Clock topic also included in this lecture.**

# **Lecture 5:**



## **What is the race condition in operating system?**

In Operating Systems (OS), a Race Condition is a situation that occurs when two or more threads or processes access a shared resource, such as a file or a variable, at the same time

## **Conflicts of serializabity of transactions:**

1. **Reading Uncommitted Data (Write-Read Conflict - WR):**

* This conflict occurs when one transaction reads data that another transaction has written but not yet committed.
* For example, Transaction A updates a record, but before it commits, Transaction B reads the same record. If Transaction A rolls back, the data read by Transaction B becomes invalid or inconsistent.

1. **Unrepeatable Read (Read-Write Conflict - RW):**

* This conflict occurs when a transaction reads data, another transaction modifies that data, and then the first transaction reads the same data again, but now it's different.
* For example, Transaction A reads a record, then Transaction B updates the same record, and finally, Transaction A reads the record again, noticing that it has changed.

1. **Lost Update (Write-Write Conflict - WW):**

* This conflict occurs when two transactions both update the same data item without knowing about each other's changes, resulting in one of the updates being lost.
* For example, Transaction A updates a record, and before it commits, Transaction B updates the same record. If Transaction A commits first, Transaction B's update gets overwritten, leading to a lost update.

## **Concurrency control in databases:**

1. **Isolation of Transactions:**

Transactions should execute independently of each other to maintain data consistency and integrity. Concurrency control ensures that transactions operate in isolation, preventing interference between them.

1. **Preventing Lost Updates:**

As mentioned earlier, lost updates occur when multiple transactions attempt to modify the same data simultaneously, resulting in one update being lost. Concurrency control mechanisms prevent this by coordinating access to shared data, ensuring that updates are not lost.

1. **Avoiding Dirty Reads:**

Dirty reads occur when a transaction reads data that has been modified by another uncommitted transaction. Concurrency control prevents such scenarios by ensuring that only committed data is accessed by transactions.

1. **Preventing Inconsistent Reads:**

Inconsistent reads happen when a transaction reads data that has been partially updated by another transaction. Concurrency control mechanisms like locking and serialization prevent inconsistent reads by ensuring that transactions access data in a consistent state.

1. **Deadlock Prevention:**

Deadlocks occur when two or more transactions are waiting for each other to release resources, resulting in a deadlock situation where none of the transactions can proceed. Concurrency control techniques, such as deadlock detection and prevention algorithms, help avoid such scenarios.

1. **Improving Concurrency:**

Concurrency control enables multiple transactions to execute concurrently, thereby improving system throughput and resource utilization. By managing access to shared resources effectively, concurrency control allows for better utilization of system resources without sacrificing data integrity.

## **Synchronization mechanism**

These 4 requirements/ condition are crucial for **preventing race conditions** and ensuring the correctness of concurrent programs.

1. **Primary Condition: Mutual Exclusion (Mutex):**

This condition ensures that only one thread or process can access a critical section of code or a shared resource at a time. It prevents multiple threads from simultaneously modifying the same data, which could lead to data corruption or inconsistent results. Achieving mutual exclusion guarantees that the integrity of shared resources is maintained.

1. **Primary Condition: Progress:**

It ensures that threads waiting to enter the critical section are not indefinitely blocked by other threads and can eventually make progress.

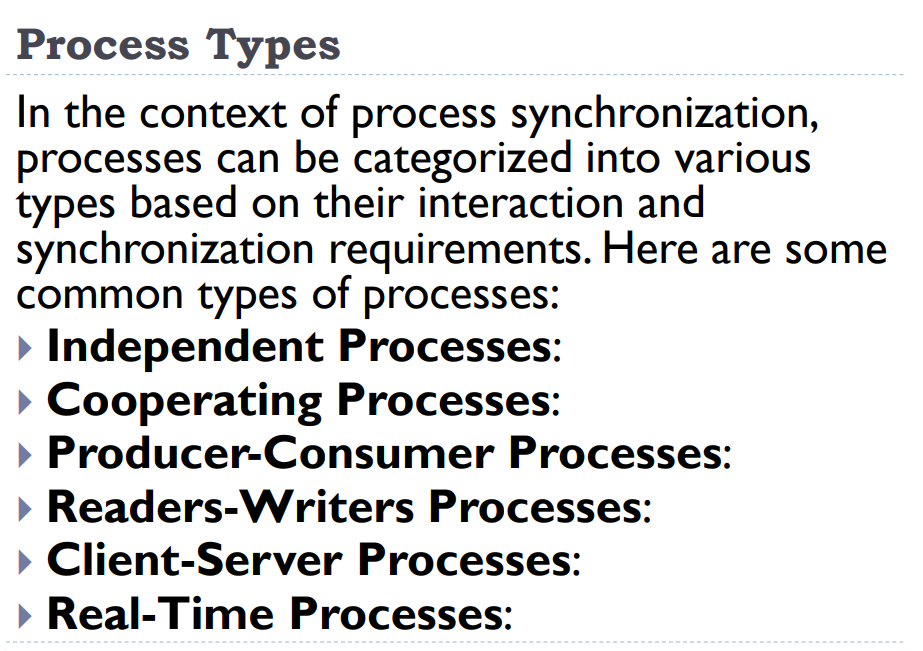
**Secondary Condition: Bounded Waiting:**

Bounded waiting ensures that there is a limit on the number of times a thread can be bypassed while waiting to enter its critical section. It prevents a situation where a thread is continuously starved or blocked from accessing the critical section indefinitely. By setting a bound on waiting, fairness is introduced to the system, ensuring that all threads have an opportunity to access shared resources over time.

**Secondary Condition: No Assumption Related to Hardware Speed:**

This condition ensures that the synchronization mechanism does not rely on assumptions about the relative speeds of different threads or processes. It means that the synchronization solution should work correctly regardless of variations in processing speed, delays in communication, or other hardware-related factors.

## **Process Types:**



**Independent Processes:**

These are processes that operate completely independently of each other. They don't communicate or share resources with other processes. Each process executes its own tasks without any dependency on other processes. In terms of synchronization, there's no need for coordination or communication between independent processes.

**Cooperating Processes:**

Unlike independent processes, cooperating processes interact with each other and may share resources or information. They work together towards a common goal, and synchronization mechanisms are often needed to ensure proper coordination. Cooperating processes might communicate through shared memory, message passing, or other inter-process communication methods.

**Producer-Consumer Processes:**

This type of process involves a producer process that produces data items and a consumer process that consumes these items. The challenge in synchronization lies in ensuring that the producer doesn't produce data too quickly for the consumer to consume, leading to data loss or inefficiency. Synchronization mechanisms such as buffers (queues) are often used to facilitate communication between producers and consumers.

**Readers-Writers Processes:**

In readers-writers synchronization, multiple processes may need access to a shared resource (such as a file or database). However, the type of access differs: readers only read the resource, while writers modify it. The challenge is to allow multiple readers to access the resource simultaneously while ensuring that writers have exclusive access, preventing data inconsistency. Synchronization mechanisms such as read and write locks are typically employed to manage access.

**Client-Server Processes:**

Client-server processes involve a client process that requests services or resources from a server process. Synchronization is important to ensure that the server can handle multiple client requests efficiently without data corruption or inconsistency. Mechanisms like semaphores, mutexes, or message queues may be used to coordinate communication and resource sharing between clients and the server.

**Real-Time Processes:**

Real-time processes have strict timing requirements, where tasks must be completed within specific time constraints to ensure proper system functioning. Synchronization in real-time systems is crucial to guarantee that tasks are executed in a timely manner and that deadlines are met. Real-time operating systems often employ specialized scheduling algorithms and synchronization mechanisms to prioritize and manage real-time tasks effectively.

## **Cooperative Process:**

1. **Variables in Cooperative Processes**

* **Shared Variables:** Processes can use global variables accessible to all, enabling communication.
* **Synchronization**: Mechanisms like mutexes, semaphores, and locks prevent race conditions.

1. **Memory in Cooperative Processes:**

* **Shared Memory Regions:** Designated memory areas accessible by multiple processes for efficient data exchange.
* **Synchronization**: Locks and semaphores manage access to prevent concurrent modifications.

1. **Code in Cooperative Processes**

* **Shared Libraries**: Loaded once, used by multiple processes to save memory and reuse code.
* **Shared Code Segments**: Allow multiple instances to run using the same code without duplication.

1. **Other System Resources in Cooperative Processes**

* **File Sharing:** Processes can read/write to the same files, with locks ensuring data integrity.
* **Devices**: Shared access to hardware like printers and storage, managed to avoid conflicts.

## **Race Condition:**

A race condition is a situation in a concurrent system where the system's behavior depends on the timing or order of execution of multiple threads or processes. This typically occurs when multiple threads or processes access shared resources simultaneously, with at least one performing a write operation. Without proper synchronization, the execution order becomes unpredictable, causing unexpected and incorrect behavior. Common manifestations of race conditions include:

* Lost Updates
* Inconsistent Reads
* Deadlocks
* Livelocks

# **Peterson Solution:**

# 

# **Lecture 6:**

## **System APIs for concurrency control:**

System APIs typically refer to platform-specific mechanisms provided by the operating system for managing concurrency, such as:

1. Thread creation,
2. Synchronization primitives (e.g., mutexes, semaphores), and
3. Inter-process communication facilities.

In the Operating System, **Mutexes and Semaphores** are kernel resources that provide synchronization services (also known as **synchronization primitives**). Synchronization is required when multiple processes are executing concurrently, to avoid conflicts between processes using shared resources.

## **Mutex:**

* It stands for Mutual Exclusion Object.
* It ensures that only one thread can access a shared resource at a time, preventing data races and ensuring data integrity.
* Mutexes are essential for preventing race conditions and ensuring thread-safe access to shared resources in multithreaded programs.

## **Semaphore:**

* A semaphore is a synchronization primitive used in concurrent programming to control access to a shared resource by multiple threads or processes.
* Semaphores maintain a count or value, which can be incremented or decremented by threads.
* Depending on the value of the semaphore, threads may either be allowed to proceed (if the count is positive) or be blocked until the count becomes positive.

**How Semaphores Work:**

**Initialization**: A semaphore starts with a certain number of "tokens" or "slots," which represents the number of resources available.

**Acquiring (Wait):** When a thread wants to use a resource, it tries to take a token from the semaphore:

If there are tokens available (the count is greater than zero), the thread takes one token (decrements the count) and continues.

If there are no tokens available (the count is zero), the thread has to wait until a token is released.

**Releasing (Signal):** When a thread finishes using a resource, it returns the token to the semaphore (increments the count), allowing another waiting thread to take it and proceed.

**Types of Semaphores:**

1. **Binary Semaphore (Mutex):**

Also known as mutexes, binary semaphores have a count of either 0 or 1.

They are typically used to control access to a single resource, ensuring that only one thread can access it at a time.

1. **Counting Semaphore**: Counting semaphores can have a count greater than 1, allowing multiple threads to access a finite pool of resources concurrently. They are useful for scenarios where multiple instances of a resource can be accessed simultaneously, up to a certain limit.

## **Amdahl's Law:**

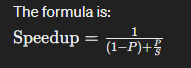
Amdahl's Law is a principle used to understand how much faster a system can get when you improve one part of it. Here’s a simple explanation:

**Overall Speedup**: Amdahl's Law tells you how much the total performance of a system improves when you make one part of it faster.

**Fraction of Work (P):** This is the part of the system that gets improved. For example, if 40% of the time is spent on a specific task, then P is 0.4.

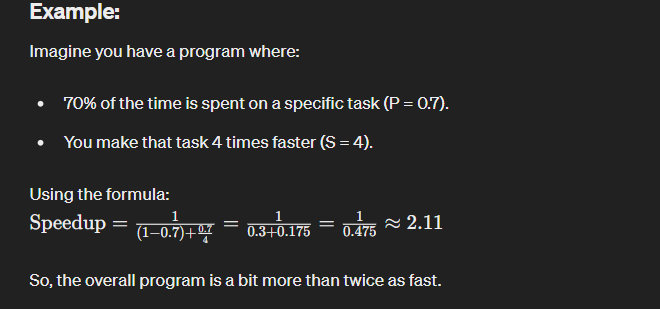
**Speedup of the Improved Part (S):** This is how much faster that part becomes. If you make that part twice as fast, S is 2.

**Formula**:



**Key Insights:**

1. **Diminishing Returns**: The more you improve one part, the less overall speedup you get from further improvements, because other parts of the system become the bottleneck.
2. **Identify Critical Parts:** Focus on optimizing the parts of the system where the most time is spent to get the best performance improvement.
3. **Balanced Optimization:** Consider the whole system, not just one part, to make meaningful performance gains.



Lec1

\* Serial vs Parallel Computing (comparison)

\* Flynn Taxonomy

SISD

SIMD

MISD

MIMD

\* Parallel vs Distributed Computing (comparison)

\* ⁠Memory Architectures

# **Lecture 7a:**

## **Definition and Key Characteristics of Distributed Systems**

**Distributed Systems** are a type of computing system in which multiple autonomous computers communicate and coordinate with each other to achieve a common goal. Here are some significant consequences and characteristics of distributed systems:

Concurrency

No global clock

Independent Failures

Motivation

1. **Concurrency**:
   * **Description**: Multiple computers (nodes) in a network can execute programs simultaneously, sharing resources such as web pages, files, databases, etc.
   * **Significance**: This parallel execution improves performance, efficiency, and resource utilization. However, coordinating these concurrent programs is complex and requires careful synchronization and communication mechanisms to ensure consistency and correct operation.
2. **No Global Clock**:
   * **Description**: Computers in a distributed system cannot perfectly synchronize their clocks due to variations in clock speeds and network latency.
   * **Significance**: Without a global clock, it's challenging to maintain a consistent global state or order of events. Therefore, programs in distributed systems coordinate their actions by exchanging messages rather than relying on synchronized clocks.
3. **Independent Failures**:
   * **Description**: In a distributed system, each node can fail independently, and network issues can cause nodes to become isolated without stopping their operation.
   * **Significance**: This leads to unique failure modes:
     + **Network Faults**: A network fault can isolate nodes, making them unable to communicate. Nodes continue operating, but may not know if the network is down or just slow.
     + **Node Failures**: A node or program can crash without immediately notifying other components. Other nodes must detect and handle these failures gracefully, ensuring that the overall system remains functional despite individual failures.
4. **Motivation**:
   * **Description**: The primary motivation for constructing and using distributed systems is the desire to share resources and achieve goals that are not possible with a single machine.
   * **Significance**: By sharing resources, distributed systems can provide better performance, scalability, and fault tolerance. They enable resource sharing across different geographical locations, support large-scale computation, and improve availability and reliability.

## **Types of Distributed Systems:**

### **1.Synchronous distributed systems:**

**Definition**: In a synchronous distributed system, there are strong assumptions about the timing of events, including message delivery times, process execution speeds, and clock synchronization.

**Characteristics**:

1. **Bounded Execution Time**:
   * The time to execute each step of a process has known lower and upper bounds.
2. **Bounded Message Delivery**:
   * Messages transmitted between channels are guaranteed to be delivered within a known bounded time.
3. **Bounded Clock Drift**:
   * Each process has a local clock whose drift rate from real time has a known bound.

**Advantages**:

It is possible to use timeouts to detect the failure of a process

Simplifies the design of certain algorithms, such as consensus algorithms.

**Disadvantages**:

Real-world networks and computers don't always run on a perfectly predictable schedule, making it tough to ensure everything happens within set time limits.

Node Failures: If one computer or process fails, it can disrupt the whole system because everything is tightly coordinated and timed.

**Conditions for Building Synchronous Systems**:

1. **Known Resource Requirements**:

If the system knows in advance how much CPU power and network capacity are needed, it can make sure there are enough resources available to meet these needs.

1. **Controlled Clock Drift**:

Clocks must be synchronized within certain bounds to ensure coordination.

**Examples**: Real-time systems where timing guarantees are crucial, such as certain types of control systems and telecommunications.

### **2.Asynchronous distributed systems:**

**Definition**: In an asynchronous distributed system, there are no assumptions about the timing of events. Message delivery times, process execution speeds, and clock synchronization can vary unpredictably.

**Characteristics**:

1. **No Bounded Execution Time**:
   * Processes can take any amount of time to complete; there are no guarantees.
2. **No Bounded Message Delivery**:
   * Messages can be delayed indefinitely or arrive at unpredictable times.
3. **No Bounded Clock Drift**:
   * Clocks on different computers can drift independently without any synchronization.

**Challenges**:

* Many problems can't be solved in an asynchronous system because timing guarantees are necessary.

For example:

**Multimedia Streaming**: Requires timely delivery of data (like video frames) to maintain quality.

Such problems need a synchronous model where timing can be controlled and guaranteed.

**Advantages**:

More realistic model for many real-world systems, as it accommodates the unpredictability of networks and processing.

More robust to network partitions and node failures.

**Disadvantages**:

More challenging to design and reason about the system’s behavior due to the lack of timing guarantees.

Certain algorithms, like consensus algorithms, become more complex and less efficient.

**Examples**: Most general-purpose distributed systems, such as the Internet, distributed databases, and cloud computing platforms.

## **Architectural Styles in Distributed Systems**

In distributed systems, processes interact with each other and take on specific roles. Here, we explore two main architectural styles based on the roles of individual processes:

1. **Client-Server**
2. **Peer-to-Peer (P2P)**

### **Client-Server Architecture**

**Overview**:

* In a client-server architecture, client processes interact with server processes, usually on separate host computers, to access shared resources managed by the servers.
* Servers can also act as clients to other servers.

**Example**:

* **Web Server**:
  + A web server provides web pages to clients (browsers).
  + It might also be a client to a local file server to fetch the web pages.
  + Web servers often rely on DNS servers to translate domain names to IP addresses.
* **Search Engines**:
  + **Web Crawlers**: Programs that run in the background, making HTTP requests to access web servers and gather information.
  + **Search Engine Server**: Responds to user queries by providing summaries of the information collected by web crawlers.
  + **Dual Role**: The search engine acts as both a server (responding to user queries) and a client (web crawlers fetching data).

**Characteristics**:

* Clients request services from servers.
* Servers manage resources and respond to client requests.
* Tasks can be independent; for example, a search engine's user query handling and web crawling tasks are independent and can run concurrently without much synchronization.

**Problem with Client-Server Model**:

* Centralization in the client-server model can lead to scalability issues as the system grows, limited by the capacity of the server and its network connections.

### **Peer-to-Peer (P2P) Architecture**

**Overview**:

* In a P2P architecture, all processes play similar roles and interact cooperatively as peers without distinguishing between clients and servers.
* All participating processes run the same program and offer the same set of interfaces to each other.

**P2P Advantages**:

* **Resource Utilization**: In a Peer-to-Peer (P2P) architecture, the system uses the combined resources of many computers to perform tasks.
* **Scalability**: Resources grow with the number of participating computers.

**Example**:

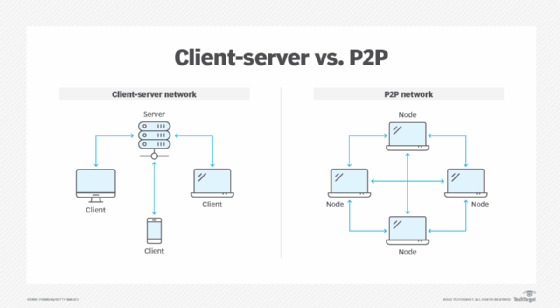
* **BitTorrent**:
  + A file-sharing system where each participant both uploads and downloads files, distributing the load and resources.

**Characteristics**:

* Data and processing are distributed across many computers.
* Each computer holds a small part of the overall data and participates equally in processing and communication tasks.
* Objects are often replicated across multiple computers for load distribution and resilience against individual computer disconnections.

**Complexity**:

* P2P systems are more complex than client-server systems due to their distributed nature and the need for mechanisms to manage data replication, load distribution, and fault tolerance.



# **Lecture 8:**

## **Challenges Facing Distributed Systems**

Distributed systems face several significant challenges in their design and use. Here are the primary challenges:

### **1. Heterogeneity**

Heterogeneity refers to the variety and differences in the components of a distributed system, which include:

* **Networks:** Different types of networks.
* **Computer Hardware:** Various types of hardware with different architectures.
* **Operating Systems:** Different operating systems with different APIs.
* **Programming Languages:** Different programming languages with unique data representations and structures.

For example, data types such as integers may be represented differently on various hardware, and operating systems might use different calls for message exchange (UNIX vs. Windows). Middleware helps manage heterogeneity by providing a uniform interface and handling differences in hardware, operating systems, and networks.

#### **Middleware**

Middleware addresses heterogeneity and supports mobile code, which can be transferred and executed on different computers. For instance:

* **Java applets** can run on various systems through the Java Virtual Machine (JVM), which needs to be implemented for each type of computer.
* **JavaScript programs** in web pages are a common form of mobile code.

### **2. Openness:**

**Explanation**: Openness in a distributed system means that the system can be extended or modified without disrupting its operation. It allows for easy addition of new services or replacement of old ones.

* **Extensibility**: A system should allow adding new features or services without major changes. For instance, you should be able to add a new service running on a different operating system.
* **Interoperability**: Different implementations of system interfaces should work together seamlessly. For example, two companies might build different systems that need to work together, and they should be able to do so without conflicts.
* **Interface Specification**: For openness, the system's interfaces need to be well-defined and published. This includes specifying how services interact and ensuring that different implementations adhere to these specifications.

**Challenges**: Ensuring that different systems can interact smoothly requires detailed and often informal specifications of service behaviors and interfaces, which can be complex.

### **3. Security:**

Security in distributed systems involves:

* **Confidentiality:** Protection against unauthorized disclosure.
* **Integrity:** Protection against unauthorized alteration.
* **Availability:** Ensuring access to resources despite interference.

Challenges include securing data transmissions (e.g., using encryption) and authenticating users (e.g., biometric techniques or verification codes).

### **4. Scalability**

A system is scalable if it remains effective even with significant increase in the number of resources and users. Scalability dimensions include:

* **Size Scalability:** Adding more users and resources.
* **Geographical Scalability:** Users and resources spread over large areas.
* **Administrative Scalability:** Managing the system across multiple administrative domains.

#### **Scalability Problems**

1. **Server Bottlenecks:** When too many users or applications use a single server, it can become overloaded and slow down.
2. **Geographical Delays**: Communication across wide-area networks (like the Internet) is much slower than within a local network (like a company’s internal network). This can slow down interactions between distant locations.
3. **Unreliable Communication:** Wide-area networks are less reliable than local networks, meaning messages can get lost or delayed more often.
4. **Location Services Issues:** Finding a service or resource in a large distributed system can be difficult because the system might need special services to locate resources quickly and accurately.
5. **Administrative Conflicts:** When a distributed system spans across different organizations or administrative domains, it can be hard to manage due to differing policies and security requirements.
6. **Performance and Reliability Issues:** In systems with many centralized components or services, performance and reliability can suffer because any failure can impact many users.

#### **Scalability Techniques**

1. **Hiding Communication Latencies**
   * When a system needs to wait for a reply from another part, it can keep doing other useful tasks in the meantime instead of just waiting.
   * Instead of stopping everything while waiting for a response, the system continues working on other things. When the reply arrives, it can handle it without having to wait idly.
2. **Distribution**
   * Breaking a big task into smaller parts and spreading them across different machines or locations.
   * Instead of having one machine do all the work, you divide the job among several machines, each handling a part of it.
   * **Example**: The Domain Name System (DNS) spreads the task of looking up website addresses across many servers so that no single server has to handle everything.
3. **Replication**
   * Making copies of important parts of the system to improve availability and performance.
   * By having several copies of a component, the system can continue working even if one copy fails. It also helps handle more requests by balancing the load among copies.

### **5. Failure Handling**

Handling failures in distributed systems is challenging because:

* **Partial Failures:** Some components may fail while others continue to function.
* **Failure Detection:** Some failures are detectable (e.g., corrupted data), while others are not (e.g., a crashed remote server).
* **Failure Masking:** Hiding or reducing the severity of failures, such as retransmitting messages or using redundant disks.
* **Tolerating Failures:** Informing users about failures and allowing them to retry later.
* **Recovery from Failures:** Designing software to recover or roll back to a consistent state after a failure.
* **Redundancy:** Using redundant components to tolerate failures, such as multiple routes in the Internet or replicated databases.

### **6. Concurrency**

Concurrency involves managing simultaneous access to shared resources:

* **Synchronization:** Ensuring data consistency through mutual exclusion techniques (e.g., semaphores).
* **Example:** In an auction system, concurrent bids must be properly recorded to maintain data integrity.

### **7. Transparency**

Transparency in distributed systems means hiding the complexities and differences of the underlying system so that users and applications experience it as a single, unified system. Here’s a breakdown of the types of transparency:

1. **Access Transparency**
   * **What It Is**: Hides the differences in how data is accessed and represented.
   * **How It Works**: Allows users to interact with resources in the same way, whether they are local or remote.
   * **Example**: You can use the same commands to access files on your computer and on a remote server.
2. **Location Transparency**
   * **What It Is**: Hides where resources are physically located in the network.
   * **How It Works**: Users or applications don’t need to know the exact location of resources to access them.
   * **Example**: You access a website by its URL without knowing which server it’s on.
3. **Concurrency Transparency**
   * **What It Is**: Allows multiple processes to use the same resource simultaneously without interfering with each other.
   * **How It Works**: Ensures that concurrent access to shared resources remains consistent and error-free.
   * **Example**: Multiple users can edit a shared document at the same time without causing conflicts.
4. **Replication Transparency**
   * **What It Is**: Hides the existence of multiple copies of a resource.
   * **How It Works**: Users interact with the resource as if there’s only one, even though there are several copies to improve reliability and performance.
   * **Example**: When accessing a file, you don’t need to know that it exists on multiple servers to ensure availability.
5. **Failure Transparency**
   * **What It Is**: Conceals faults and ensures that the system continues to function despite failures.
   * **How It Works**: Manages failures in a way that users and applications are unaware of the issues.
   * **Example**: If a server crashes, another server takes over without affecting your access to the service.
6. **Migration Transparency**
   * **What It Is**: Allows resources to be moved from one location to another without affecting how they are accessed.
   * **How It Works**: Users can continue accessing resources normally, even if they have moved to a different location.
   * **Example**: A file moved to a different server doesn’t require users to update their access methods.
7. **Performance Transparency**
   * **What It Is**: Ensures that performance remains consistent even as the system is reconfigured or scaled.
   * **How It Works**: Adjusts the system’s performance to handle varying loads without affecting how users interact with it.
   * **Example**: A website maintains its speed even as more users visit it by scaling resources appropriately.
8. **Scaling Transparency**
   * **What It Is**: Allows the system to grow in size without changing how applications or users interact with it.
   * **How It Works**: Supports system expansion seamlessly, so users don’t notice changes as the system scales.
   * **Example**: A video streaming service can handle more users without requiring them to adjust their settings or experience interruptions.

# **Lecture 9:**

### **Inter-process Communication on the Internet**

1. **Types of Communication**
   * **Datagram Communication** (e.g., UDP)
   * **Stream Communication** (e.g., TCP)

### **UDP (Universal Datagram Protocol)**

**Features**

1. Independent packets of data.
2. Packets sent without establishing a connection.
3. Each packet is delivered individually without ongoing connection.
4. Packets may follow different routes to the destination.
5. No guarantee of delivery; packets may be lost or arrive out of order.
6. UDP adds no additional reliability mechanism except the checksum which is optional.
7. It has a short header that includes the source and destination port numbers, a length field and a checksum.

**Use of UDP:** Acceptable for applications where occasional message loss is acceptable.

**Advantages**: Lower overhead compared to TCP (no connection setup or guaranteed delivery).

**Overheads**: Avoids extra state information, additional messages, and transmission latency.

### **Sockets**

#### **Definition**

A software interface, in a process, for sending and receiving messages from the network. Both forms of communication (UDP and TCP) use the socket abstraction, which provides an endpoint for communication between processes. Inter-process communication consists of transmitting a message between a socket in one process and a socket in another process.

#### **2. Characteristics:**

For a process to receive messages, its socket must be bound to a local port and one of the Internet addresses of the computer on which it runs.

Computers have many ports (up to 65,536) for local processes; processes cannot share ports but can use multiple.

Messages sent to a particular Internet address and port number can be received only by a process whose socket is associated with that Internet address and port number.

Processes may use the same socket for sending and receiving messages.

Each socket is associated with a particular protocol – either UDP or TCP.

#### **3. How Sockets Work**

1. **Creating a Socket:** A process creates a socket to establish a communication channel.
2. **Binding:** Binding a socket means associating it with a specific port number and an IP address.
3. **Listening and Accepting Connections:** In a server setup, the socket listens for incoming connection requests.
4. **Connecting:**: A client socket connects to the server’s socket using the server’s IP address and port number.
5. **Sending and Receiving Data:** Once the connection is established, data can be sent and received through the socket.
6. **Closing the Socket:** After communication is done, the socket is closed to free up resources.

#### **4. Types of Sockets**

1. **UDP (User Datagram Protocol) Sockets:**
   * **Characteristics**: Simple, connectionless, and fast.
   * **Usage**: Good for applications where speed is crucial and occasional data loss is acceptable (e.g., streaming media).
2. **TCP (Transmission Control Protocol) Sockets:**
   * **Characteristics**: Reliable, connection-oriented, and ensures data arrives in order.
   * **Usage**: Suitable for applications requiring reliable data transfer (e.g., web browsing or file transfers).

### **Issues with Datagram Communication**

1. **Message Size**

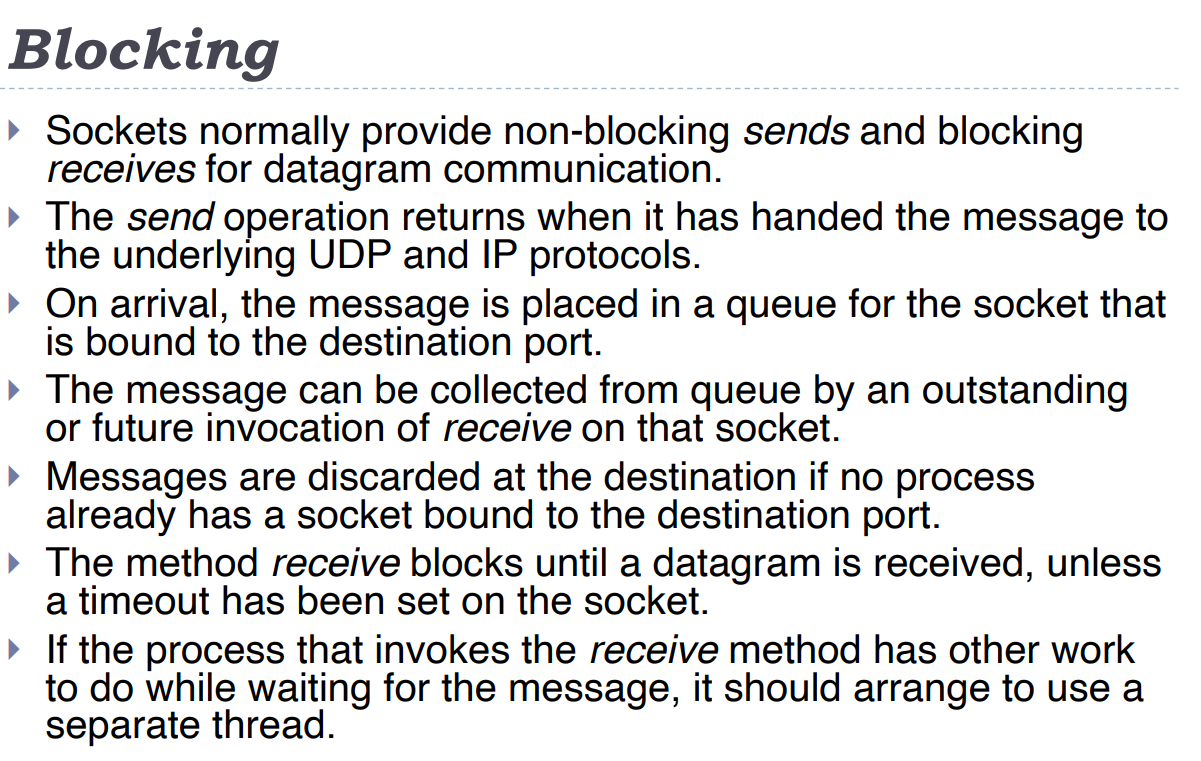
The underlying IP protocol allows packet size up to 2^16 (64k) bytes, which includes the headers as well as message.

However, most environments impose a size restriction of 8 kilobytes.

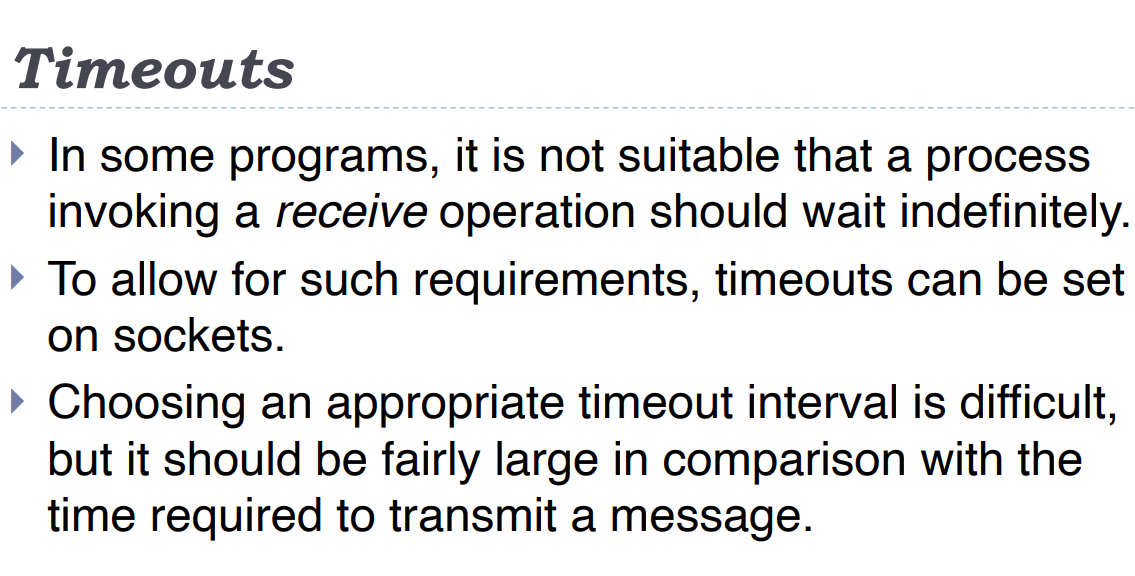
Any application requiring messages larger than maximum must fragment them into chunks of that size.

Generally, an application, e.g. DNS, will use size not excessively large.

1. **Blocking:**

****

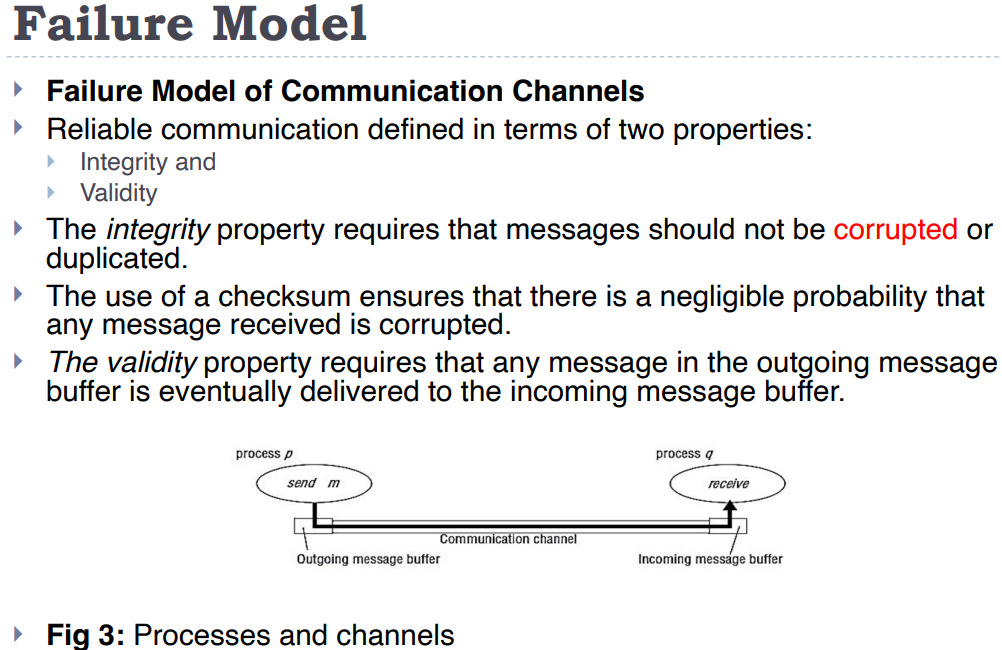
1. **Timeouts**

****

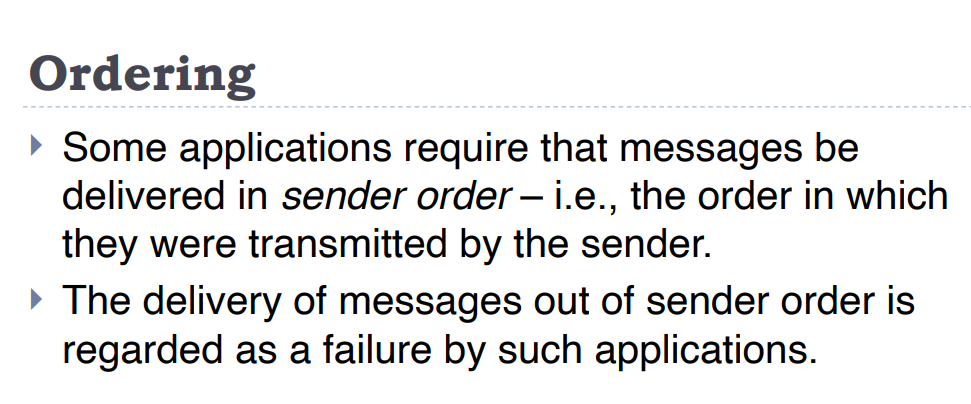
#### **4. Receive from Any:**

* **Receiving Messages:**
  + When you use the receive method on a socket, it doesn't care where the message is coming from. It just receives the message.
  + Think of it like opening your mailbox: you don’t know who will send you a letter until you open it.
* **Getting Sender Information:**
  + After receiving a message, the receive method tells you where the message came from. It provides the sender’s IP address and port number.
  + This is like looking at the return address on a letter to see who sent it.
* **Connecting to Specific Address:**
  + You can also set up your socket to only send and receive messages from a specific IP address and port number.

### **Failure Model:**



### **Ordering:**



# **Lecture 10a:**

## **Characteristics of blocking and non-blocking communication:**

**Blocking:**

**Synchronization**:

* The sender and receiver wait for each other, ensuring that messages are properly sent and received in sequence.

**Simple Programming Model**:

* Easier to implement and understand because operations complete in a predictable order.

**Potential Deadlocks**:

* Can lead to deadlocks if processes are waiting indefinitely for each other to complete communication.

**Resource Utilization**:

* Can be less efficient as processes may be idle while waiting for communication to complete.

**Non- Blocking:**

**Asynchronous**:

* Communication operations do not require the sender or receiver to wait for completion.

**Overlap of Computation and Communication**:

* Processes can continue executing other tasks while waiting for the communication to finish.

**Complex Programming Model**:

* Requires careful handling to manage the concurrent execution and communication, making the code more complex.

**Avoidance of Deadlocks**:

* Less prone to deadlocks since processes are not waiting for communication operations to complete.

## **Types of communication channel**

### **1. Direct vs. Indirect Communication**

**Direct Communication**:

* **Definition**: Processes communicate directly with each other, typically specifying the exact recipient of a message.
* **Example**: If Process A wants to send a message to Process B, it sends the message directly to Process B's address or identifier.

**Indirect Communication**:

* **Definition**: Processes communicate through an intermediary, like a mailbox or message queue, rather than directly.
* **Example**: Process A sends a message to a shared message queue, and Process B later retrieves the message from the queue.

### **2. Synchronous vs. Asynchronous Communication**

**Synchronous Communication**:

* **Definition**: The sender and receiver must be synchronized. The sender waits until the receiver has received the message before continuing.
* **Example**: A process sends a message and waits for an acknowledgment before proceeding.

**Asynchronous Communication**:

* **Definition**: The sender and receiver do not need to be synchronized. The sender can continue its execution without waiting for the receiver to process the message.
* **Example**: A process sends a message and continues to execute while the receiver processes the message at a later time.

### **3. Explicit vs. Automatic Communication**

**Explicit Communication**:

* **Definition**: The communication protocol is defined explicitly by the processes. The processes manually manage the sending and receiving of messages.
* **Example**: Processes explicitly specify the details of how and when messages are sent and received, often using specific IPC functions or APIs.

**Automatic Communication**:

* **Definition**: The communication is managed automatically by the system or middleware. The processes do not need to handle the details of message passing directly.
* **Example**: A system handles message delivery and receipt transparently, often through higher-level abstractions or frameworks.

## **Scheduling**

In MPI (Message Passing Interface), scheduling refers to how communication operations (sending and receiving messages) are managed and executed. MPI supports two main scheduling approaches: static and dynamic.

### **1. Static Scheduling**

**Definition**:

* **Static Scheduling** involves determining and fixing communication operations before the program starts executing.

**Characteristics**:

* **Predefined Operations**: The sequence and details of communication operations are set up in advance.
* **Predictable**: The behavior of message passing is predictable since the schedule is established before execution.
* **Less Flexibility**: Less adaptable to changing conditions during execution because the communication plan is fixed.

**Example**:

* A program where all messages to be sent and received are specified in the code before running. The program will follow this fixed schedule throughout its execution.

### **2. Dynamic Scheduling**

**Definition**:

* **Dynamic Scheduling** involves determining and adjusting communication operations during program execution based on runtime conditions.

**Characteristics**:

* **Runtime Decision**: The schedule of communication operations can change based on current conditions, such as load or network status.
* **Adaptability**: More flexible and can adapt to varying execution scenarios, optimizing performance based on real-time information.
* **Complexity**: More complex to implement because it requires mechanisms to manage and adjust the schedule dynamically.

**Example**:

* A program where the sequence of sending and receiving messages is decided during execution, allowing the system to adapt to real-time changes or performance issues.

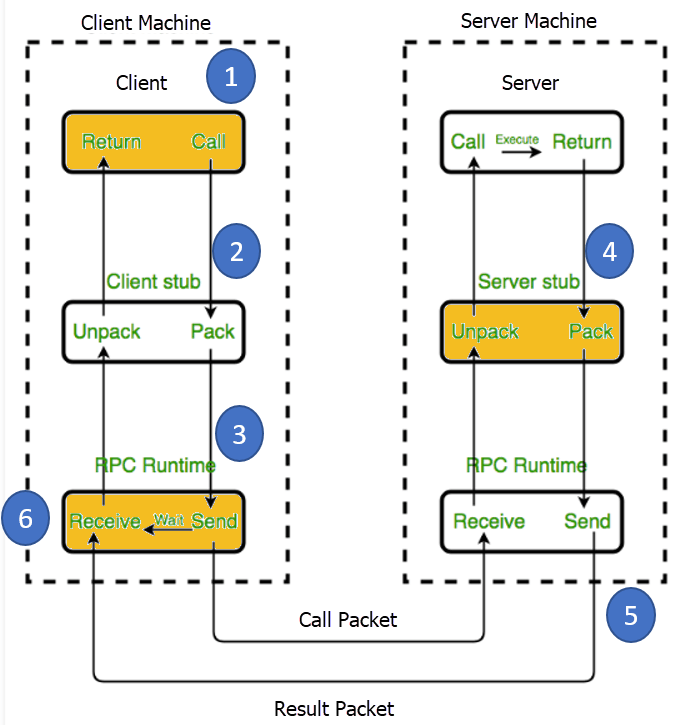
# **Lecture 11 & 16:**

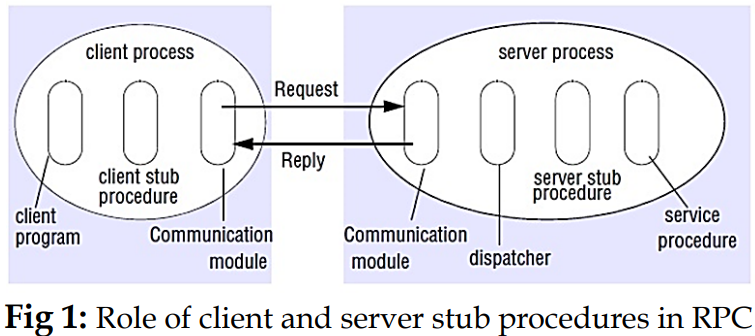
## **Remote Procedure call:**

A remote procedure call (RPC) is a protocol that allows a computer program to cause a subroutine or procedure to execute in another address space (commonly on another computer on a shared network) without the programmer explicitly coding the details for this remote interaction.

## **RPC Architecture Components**

1. **Client**: This is the computer or program that initiates the request for a service or task.
2. **Client Stub**: This acts as a bridge between the client and the server. It takes the client's request, marshalls/packs it up, and sends it to the server.
3. **RPC Runtime**: This is the system component that manages the communication between the client and server. It handles message transmission, retries if messages are lost, acknowledgments, routing, and encryption.
4. **Server Stub**: This is the counterpart to the client stub on the server side. It unpacks the client's request, calls the appropriate service on the server, and then packs the response to send back to the client.
5. **Server**: This is the computer or program that provides the service or performs the task requested by the client.





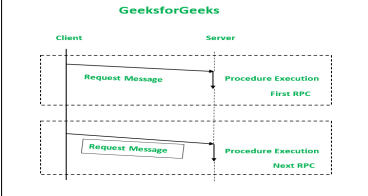
## **Steps of RPC:**

1. **Client and Initial Setup**:
   * **Step 1**: The client, client stub, and an instance of the RPC runtime execute on the client machine.
   * **Components**:
     + The **client** is the application that initiates the RPC.
     + The **client stub** is a piece of code that represents the remote function on the client side.
     + The **RPC runtime** manages the communication between the client and server.
2. **Client Stub Process**:
   * The client starts a client stub process by passing parameters as it would for a regular local procedure call. The client stub resides within the client’s own address space and communicates with the local RPC runtime to forward the request to the server stub.
3. **RPC Runtime Management**:
   * **Step 3**: The RPC is accessed by the user through a regular local procedure call. The RPC runtime handles the transmission of messages across the network between the client and server. It manages tasks such as message retransmission (if a message is lost), acknowledgment, routing, and encryption.
4. **Server Processing**:
   * The server dispatcher receives the request message and selects the appropriate server stub based on the procedure identifier.
   * The server stub unmarshals the arguments and calls the corresponding service procedure.
   * The service procedure executes and returns the results.
   * The server stub marshals the results into a reply message and sends it back to the client.
5. **Message Transmission**:
   * The transport layer on the server side sends the result message back to the transport layer on the client side. The client transport layer then forwards the message to the client stub.
6. **Client Stub Finalization**:
   * The client stub demarshals (unpacks) the return parameters from the message received. The execution process then returns the results to the caller, completing the RPC process.

## **The Request Protocol**

Also known as the R Protocol, is a streamlined communication protocol used in Remote Procedure Call (RPC) systems. It is designed for scenarios where the calling procedure does not need to receive a return value or confirmation from the called procedure. Here’s an overview of the key features and use cases of the R Protocol:

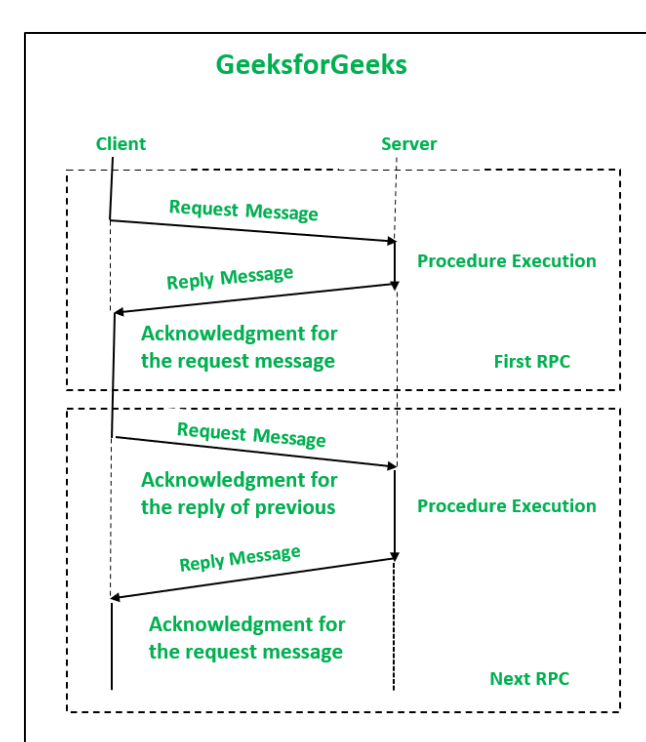
1. **Basic Functionality**:
   * **One-Way Communication**: The R Protocol is used when a request is made from the client (calling procedure) to the server (called procedure) without expecting a return value or confirmation of execution.
   * **Single Message**: Only one message is sent from the client to the server, and no reply or acknowledgment is expected.
2. **May-Be Call Semantics**:
   * **No Retransmission**: The protocol does not require retransmission of request packets, making it suitable for situations where occasional message loss can be tolerated.
   * **Proceed Without Waiting**: After sending the request message, the client can immediately proceed with subsequent tasks or requests without waiting for a response from the server.
3. **Asynchronous RPC**:
   * **Enhanced Performance**: Asynchronous Remote Procedure Call (RPC) utilizes the R Protocol to improve the combined performance of the client and server.
   * **No Waiting for Replies**: The client does not wait for a reply from the server, and the server does not send one. This allows both the client and server to continue processing other tasks independently.
   * **Failure Handling**: In the event of communication failure, the RPC runtime does not retry the request, which simplifies error handling and reduces overhead.
4. **Transport Protocols**:
   * **TCP vs. UDP**: While TCP is a connection-oriented protocol and ensures message delivery, it is generally preferred for reliable communication. However, the R Protocol can be used with UDP, which is a connectionless protocol that does not guarantee delivery but has lower overhead.



## **Request/Reply Protocol (RR Protocol)**

1. **Basic Functionality**:
   * Involves two-way communication where the client sends a request, and the server replies.
   * Suitable for simple RPCs with parameters and result values in a single packet.
2. **Implicit Acknowledgment**:
   * The server's reply acts as an acknowledgment (ACK) for the client's request.
   * The client's next request acts as an acknowledgment of the server's previous reply.
3. **Timeout and Retransmission**:
   * If the client does not receive a reply within a predetermined timeout period, it retransmits the request.
   * This mechanism handles lost messages and ensures reliable communication.
4. **Exactly-Once Semantics**:
   * Servers use a reply cache to store responses and filter duplicate requests.
   * If a duplicate request is received, the server retransmits the stored reply without reprocessing the request.
5. **At-Least-Once Semantics**:
   * Used when duplicate message filtering is unavailable, ensuring the client resends requests until a reply is received.

**Drawback:** Responses are held in the server’s reply cache to ensure delivery. If a reply is not delivered, it might get lost, causing the client to not receive the necessary response.



**The Request/Reply/Acknowledgement-Reply (RRA)**

The Request/Reply/Acknowledgement-Reply (RRA) Protocol is designed to enhance the reliability of communication in distributed systems by addressing the shortcomings of the simpler Request/Reply (RR) Protocol. Here's a breakdown of how it works and its key features:

* **Enhanced Functionality:** The RRA protocol extends the RR protocol by adding an acknowledgment step from the client.
* **Client Acknowledgment:** After receiving a reply from the server, the client sends back an acknowledgment to the server.
* **Server Cache Management:** The server only removes the reply from its cache after receiving the acknowledgment from the client. This ensures that the client has received the reply.
* **Handling Lost Acknowledgments:** Since acknowledgment messages can also be lost, the RRA protocol uses unique, ordered message identities. This helps in tracking which acknowledgments have been sent and received.
* **Message Identity Tracking:** The protocol keeps track of the series of acknowledgments sent to ensure the correct order and completeness.

By requiring an acknowledgment from the client, the RRA protocol ensures that the server can confirm the client has received the reply before deleting it from its cache. This mechanism helps in achieving reliable and exactly-once delivery semantics even in the presence of message losses.

****

## **Steps in the RPC Process**

1. **Client Initiates Request**:
   * The client initiates the process by calling a local procedure and passing the necessary parameters.
   * The client stub, which resides within the client's address space, takes the request and prepares it for transmission.
2. **Client Stub Prepares Request**:
   * The client stub marshals (packs) the parameters into a message.
   * The client stub then sends this message to the local RPC runtime on the client machine.
3. **RPC Runtime Manages Communication**:
   * The RPC runtime on the client side transmits the message across the network to the server.
   * The RPC runtime handles network communication tasks, such as retransmission in case of message loss, acknowledgment of message receipt, routing, and encryption to ensure secure transmission.
4. **Server Stub Processes Request**:
   * The message arrives at the server, where the server stub receives it.
   * The server stub unmarshals (unpacks) the parameters from the message.
   * The server stub then calls the appropriate procedure on the server to perform the requested task.
5. **Server Stub Sends Response**:
   * Upon completion of the task, the server procedure returns the result to the server stub.
   * The server stub marshals (packs) the return values into a message.
   * This message is sent back to the transport layer of the RPC runtime on the server side.
6. **RPC Runtime Returns Result**:
   * The RPC runtime on the server side transmits the response message back to the client's transport layer.
   * The client's RPC runtime receives the message and passes it back to the client stub.
7. **Client Stub Receives Response**:
   * The client stub unmarshals (unpacks) the return values from the message.
   * The client stub returns the results to the client, completing the execution process.

## **Characteristics of RPC**

* **Remote Location**:

The function you call is located in another process, usually on a different computer.

* **Separate Spaces**:

The client and server do not share memory; they work in separate environments.

* **Parameter Passing**:

Only the values of parameters are sent, not references or pointers.

* **Server Environment**:

The function runs on the server's system, not on the client's.

* **No Caller Access**:

The function on the server cannot directly access the client's data or environment.

## **Features of RPC**

* **Simple to Use**:

Calling a remote function looks just like calling a local one.

* **Predictable Behavior**:

You can expect the remote function to work in a consistent and understandable way.

* **Clear Interface**:

Both the client and server know exactly how to communicate with each other.

* **Flexible Communication**:

It works for communication between programs on the same machine or on different machines.

## **RPCs that involve long-duration calls or large gaps between calls:**

1. **Client Probes Server Regularly**:
   * After the client sends a request to the server, it keeps checking if the server is still working on it by sending "are you there?" messages (probe packets).
   * The server must reply to these probe packets to let the client know it's still working on the request.
   * If the client doesn't get a response, it informs the user that there might be a communication problem.
2. **Server Sends Regular Acknowledgements**:
   * If the server takes a long time to process the request, it sends regular "I’m still working on it" messages (acknowledgements) back to the client.
   * These acknowledgements reassure the client that the server is still processing the request and hasn't crashed.
   * If the client doesn’t receive these acknowledgements within a certain time, it concludes that there might be a problem, such as the server crashing or a communication failure, and alerts the user.

## **RPCs that involve parameters/arguments and/or result in values that are too large to fit in a single:**

#### **Handling Long Messages**

* **Many Physical RPCs for One Logical RPC**:
  + When a message is too long, it's broken down into several smaller messages.
  + Each small message is sent as a separate RPC, each fitting into a single datagram packet.
  + This method is inefficient because each small message incurs overhead (extra processing), regardless of the amount of data.

#### **Multidatagram Messages**

* **Breaking Down Long Messages**:
  + Another way to handle long messages is to divide them into multiple packets.
  + These packets are sent together in a group (multidatagram message).
  + Instead of acknowledging each packet separately, the server uses a single acknowledgement for all packets, improving communication performance.

## **Advantages of RPC**

1. **Ease of Communication**:
   * Helps clients communicate with servers using familiar procedure calls in high-level languages.
2. **Local Procedure Call Model**:
   * Modeled on the local procedure call, but the called procedure runs on a different process, often on a different computer.
3. **Minimal Code Changes**:
   * Reduces the need to re-write and re-develop code.
4. **Versatility**:
   * Can be used for both distributed and local environments.
5. **Provides Abstraction**:
   * Keeps the message-passing nature of network communication hidden from the user.
6. **Distributed Environment Support**:
   * Allows applications to be used in a distributed environment as well as a local one.

## **Disadvantages of RPC**

1. **Parameter Passing Limitations**:
   * Passes parameters by value only; pointers are not allowed.
2. **Higher Overheads**:
   * Remote procedure call (and return) time can be significantly higher than local procedure calls, leading to more overhead.
3. **Vulnerability to Failures**:
   * Highly vulnerable to failures due to reliance on the communication system, another machine, and another process.
4. **Lack of Standardization**:
   * Can be implemented in various ways, leading to a lack of standardization.
5. **Limited Flexibility**:
   * Offers limited flexibility for hardware architecture as it is mostly interaction-based.
6. **Increased Cost**:
   * The cost of the process is increased due to the remote procedure call.

## 

## **RPC Call Semantics**

RPC allows you to execute a procedure on a remote server as if it were a local procedure call. However, because of network communication, there are different ways to handle the reliability and semantics of these calls:

1. **Maybe Semantics**:
   * **What It Means**: The procedure might run, but there's no guarantee. It could run once or not at all.
   * **Issues**:
     + **Omission Failures**: If the request or reply message is lost, the procedure might not run.
     + **Crash Failures**: If the server crashes, the procedure might not run.
     + **Asynchronous Delays**: The result might arrive after the expected time.
   * **Use Case**: Suitable for situations where occasional failures are acceptable.
2. **At-Least-Once Semantics**:
   * **What It Means**: The procedure will run at least once. If the server doesn’t reply, the client will retry sending the request.
   * **Issues**:
     + **Arbitrary Failures**: The procedure might run multiple times, which can cause issues if the operation should only be performed once (e.g., adding money to an account).
     + **Crash Failures**: If the server crashes, the request might need to be retried.
   * **Use Case**: Useful when it’s crucial to execute the procedure at least once, and the operations can handle being executed multiple times.
3. **At-Most-Once Semantics**:
   * **What It Means**: The procedure will execute at most once. It uses fault-tolerance measures to ensure that if a request is retried, the procedure won’t execute multiple times.
   * **Issues**:
     + **Retries and Timeouts**: It handles retries and ensures that a procedure doesn’t run more than once even if the request is retried.
   * **Use Case**: Ideal when you need a guarantee that the procedure runs exactly once and no more.

## **Transparency in RPC**

Transparency means that remote procedure calls should feel as seamless and straightforward as local procedure calls. The goal is to make remote interactions as transparent as possible:

1. **Location Transparency**:
   * You shouldn’t need to know where the server is physically located. The process of calling it should be the same as if it were on the same machine.
2. **Access Transparency**:
   * Accessing remote procedures should be done in the same way as accessing local procedures, hiding the complexity of the network and remote execution.
3. **Challenges**:
   * **Latency**: Remote calls are much slower than local calls due to network delays.
   * **Fault Tolerance**: Handling errors and ensuring that remote calls don’t fail unexpectedly.
   * **Parameter Passing**: Remote calls usually don't support passing parameters by reference, which can make them less straightforward than local calls.
4. **Design Choices**:
   * RPC systems often allow developers to choose how transparent they want the remote calls to be. This might involve making remote operations explicit or providing facilities for specifying how RPCs handle failures and retries.

## **Design issues for RPC:**

I. Style of programming promoted by RPC – programming with interfaces;

1. Server var not access from client
2. Parameter value
3. Address pass nhi

II.Call semantics associated with RPC;

Maybe

At least once

At most once

III.Key issue of transparency and how it relates to RPCs.

# 

# **Lecture 12:**

## **1. What is External Data Representation?**

External data representation (EDR) refers to the standardized format used to encode data structures and primitive values for transmission over a network or for storage. This format ensures that data can be correctly interpreted and reconstructed by different systems, which may have different internal representations.

EDR allows complex data structures (like objects, arrays, and nested structures) to be "flattened" into a byte stream for transmission and then "rebuilt" on the receiving end.

## **2. Why is External Data Representation Needed?**

**Heterogeneous Systems**: Different systems (e.g., different operating systems, hardware architectures) may represent data types (like integers, floating-point numbers, and characters) differently. EDR provides a common format that all systems can understand.

**Communication**: When data needs to be transmitted over a network between different systems or stored in a format that other systems can read, EDR is crucial.

**Consistency**: EDR ensures that data remains consistent and uncorrupted when moving between systems or storage formats.

**Different Orderings for Integers**:

* + **Big-endian**: Most significant byte comes first.
  + **Little-endian**: Least significant byte comes first.

**Different Representations for Floating-Point Numbers**.

**Different Character Codes**:

* + **ASCII**: 1 byte per character.
  + **Unicode**: 2 bytes per character.

## **3. Methods for Exchanging Binary Data Values**

**Agreed External Format**:

* + Convert values to a standard format before transmission.
  + Convert back to local format upon receipt.
  + If both computers are the same type, conversion might be skipped.

**Sender's Format with Format Indication**:

* + Send values in the sender's format along with an indication of the format used.
  + The recipient converts the values if necessary.

## **4. Marshalling and Unmarshalling**

* **Marshalling**: Converting structured data and primitive values into an external data representation (flattening).
* **Unmarshalling**: Rebuilding the data structures and primitive values from their external representation.

## **5. Approaches to External Data Representation and Marshalling**

1. **CORBA’s Common Data Representation (CDR)**:
   * For representing structured and primitive types in CORBA.
   * Usable by various programming languages.
2. **Java’s Object Serialization**:
   * For flattening and representing any single object or tree of objects in Java.
   * Usable only by Java..
3. **XML (Extensible Markup Language)**:
   * Textual format for representing structured data.
   * Originally for documents but now also used for data exchange in web services.

## **6. Design Issues in Marshalling Methods**

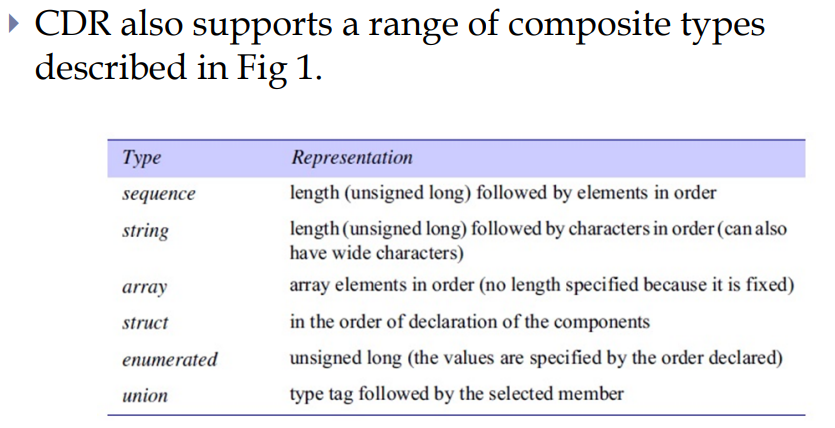
1. **Who Performs Marshalling/Unmarshalling?**
   * Typically done by middleware, not the application programmer.
   * Even for XML, tools are available to handle it automatically.
2. **Compactness**:
   * Binary representations (like CORBA CDR) are more compact.
   * Textual representations (like XML) are generally larger.
3. **Type Information**:
   * CORBA CDR includes only the values, not types.
   * Java serialization and XML include type information in different ways.

## **7. CORBA’s Common Data Representation (CDR)**

CDR can represent all of the data types that can be used as arguments and return values in remote invocations in CORBA.

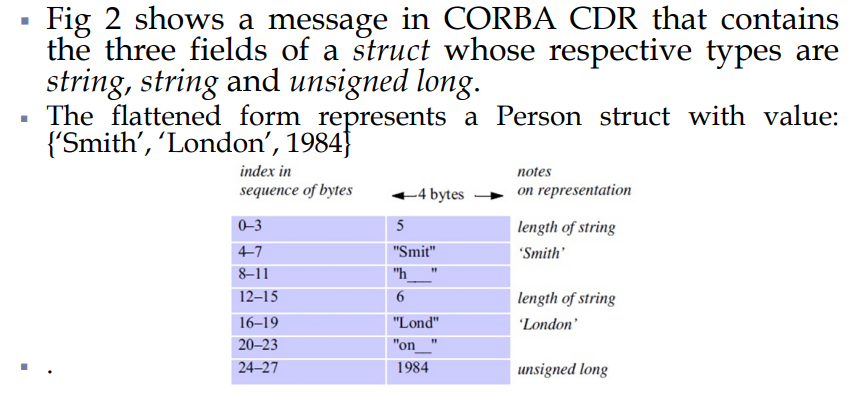
These consist of primitive as well as composite types.

The 15 primitive types included are o short (16-bit), long (32-bit), unsigned short, unsigned long, o float (32-bit), double (64-bit), char, boolean (TRUE, FALSE), o octet (8-bit), and any (which can represent any basic or constructed type);



CDR defines a representation for both **big-endian** and **little-endian** orderings.

Each primitive value is placed at an index in the sequence of bytes according to its size.



CORBA CDR can represent any data structure that can be composed from the primitive and constructed types, **but without using pointers**.

The types of the data structures and the types of the basic data items are described in **CORBA IDL**.

## **8. Example of CORBA CDR**

* **Struct Representation**:
  + Example: A Person struct with name, place, and year.
  + Flattened form: {"Smith", "London", 1984}

## **9. Marshalling in CORBA**

* **Automatic Generation**:
  + Marshalling operations are generated from the type definitions in CORBA IDL (Interface Definition Language).

## **Marshalling and Unmarshalling:**

### **Marshalling**:

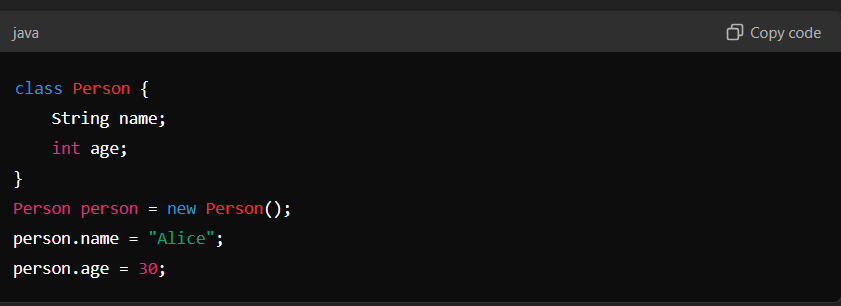
Marshalling is the process of transforming the memory representation of an object or data structure into a format suitable for storage or transmission. This process converts complex data types and structures into a byte stream, making it possible to send the data over a network or store it in a file.

**Key Points:**

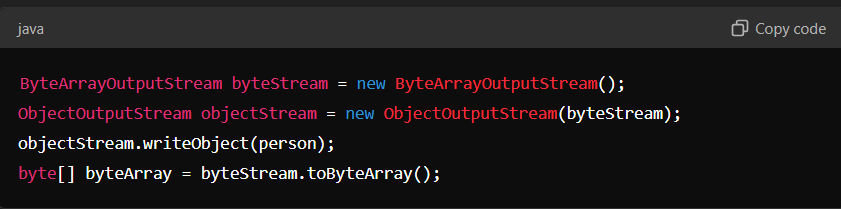
* **Purpose**: Prepare data for transmission or storage.
* **Process**: Converts objects or data structures into a serialized format (a byte stream).
* **Usage**: Commonly used in remote procedure calls (RPCs), remote method invocations (RMIs), and data storage.

**Example:**

1. **Data Structure**: Suppose we have a Person object with a name and age.



1. **Marshalling**: Convert the Person object into a byte stream for transmission.



Now, byteArray contains the serialized form of the Person object.

### **Unmarshalling**

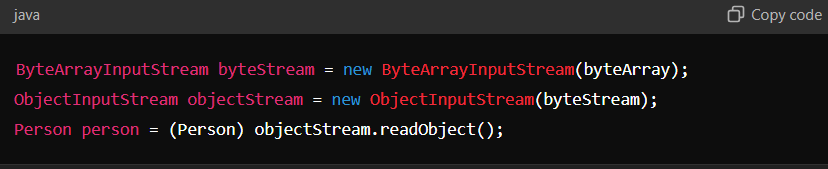
**Unmarshalling** is the reverse process of marshalling. It involves converting the byte stream back into the original data structure or object. This allows the received data to be used as if it were originally created on the receiving system.

**Key Points:**

* **Purpose**: Reconstruct data from a transmitted or stored byte stream.
* **Process**: Converts a serialized byte stream back into the original data structure or object.
* **Usage**: Used when receiving data from a network or reading from a stored file.

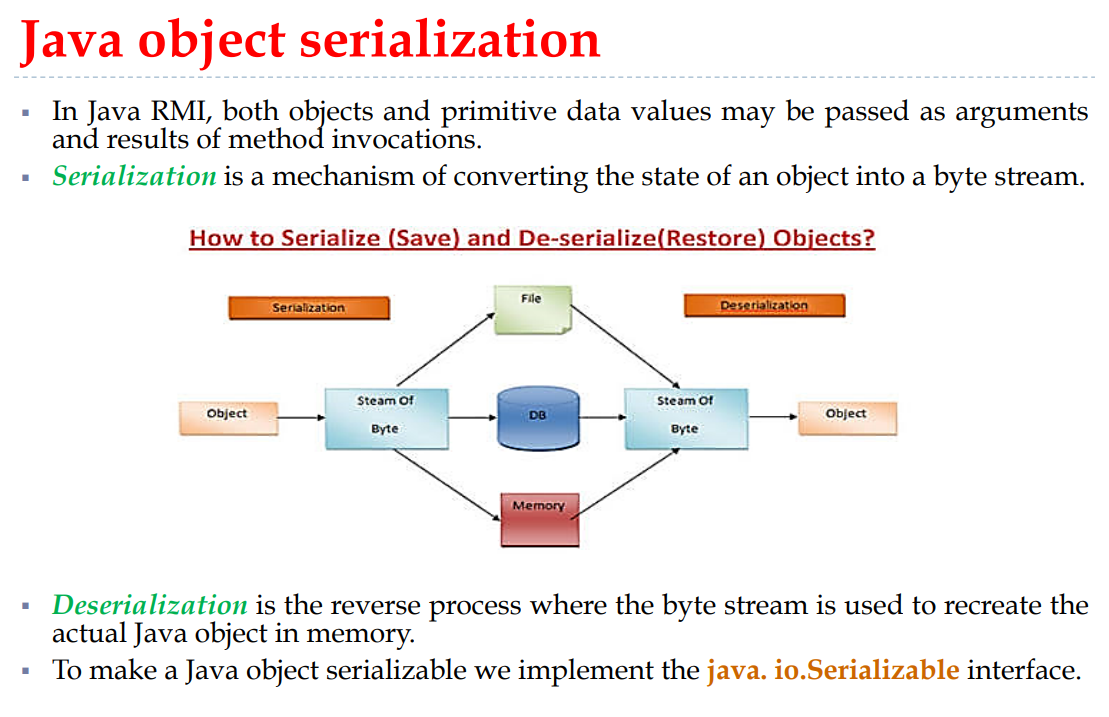
**Example:**

1. **Byte Stream**: Assume we have received the byteArray from the marshalling example above.
2. **Unmarshalling**: Convert the byte stream back into a Person object.



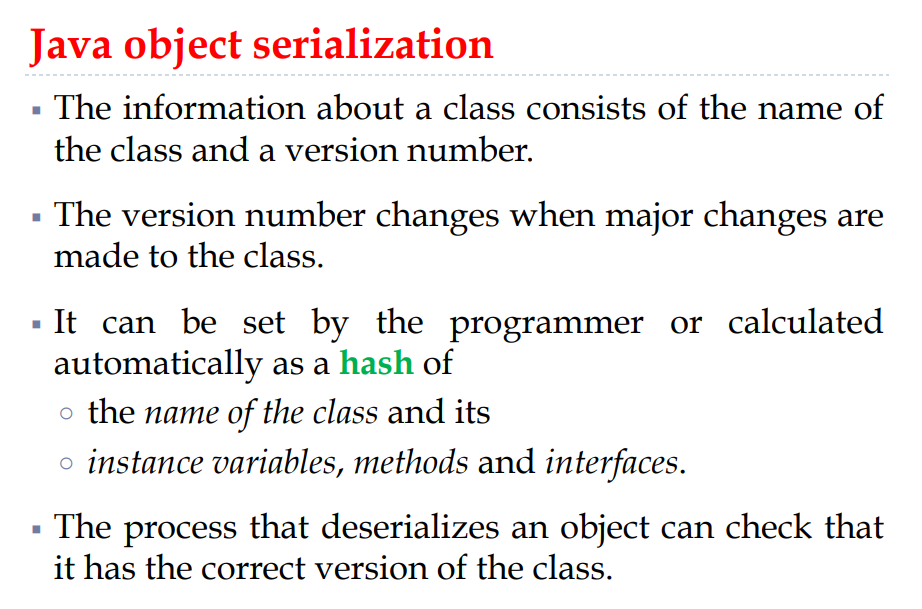
Now, person is a Person object reconstructed from the byte stream.

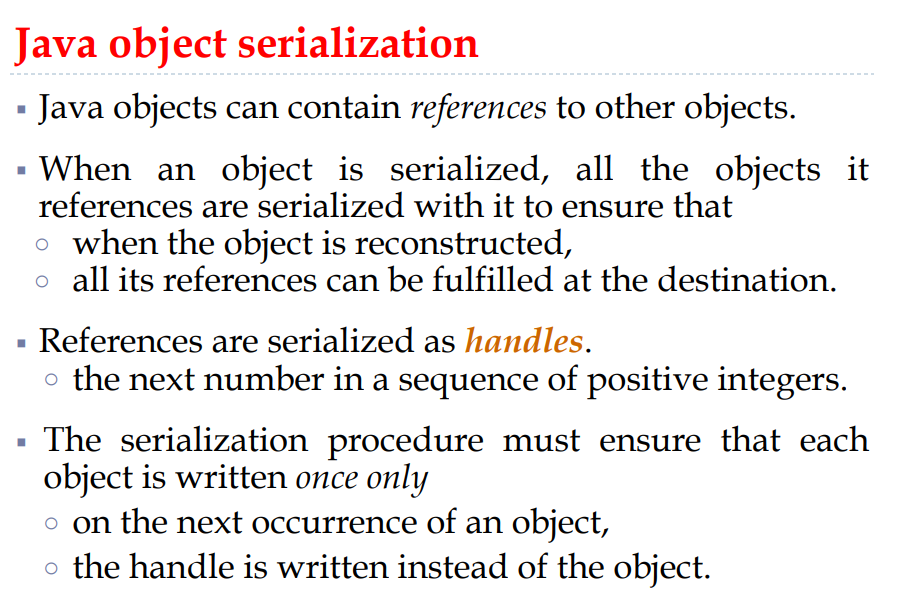
# **Lecture 13:**

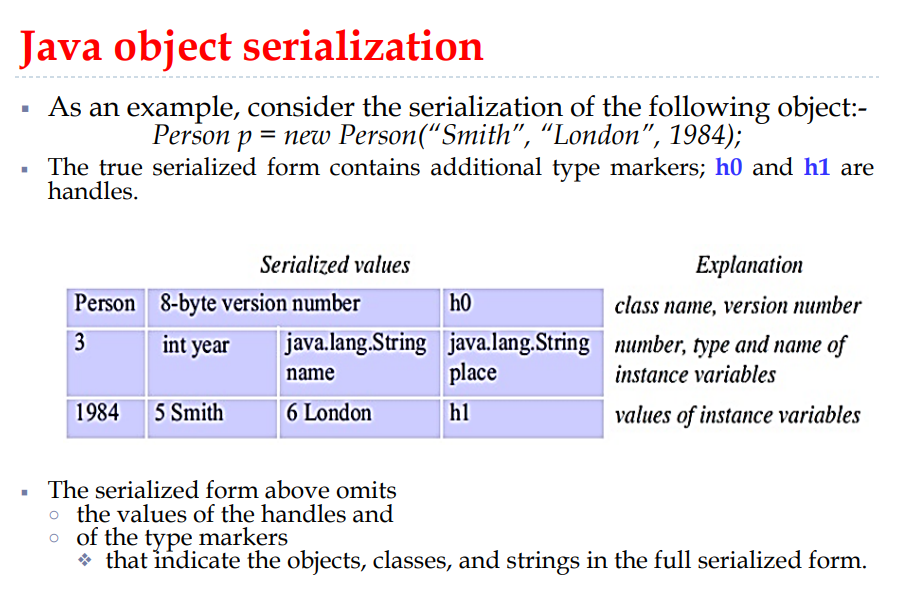


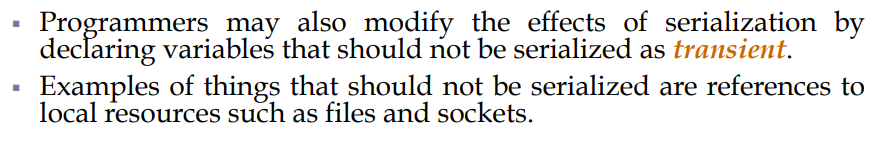
It is assumed that the process that does the deserialization has no prior knowledge of the types of the objects in the serialized form.

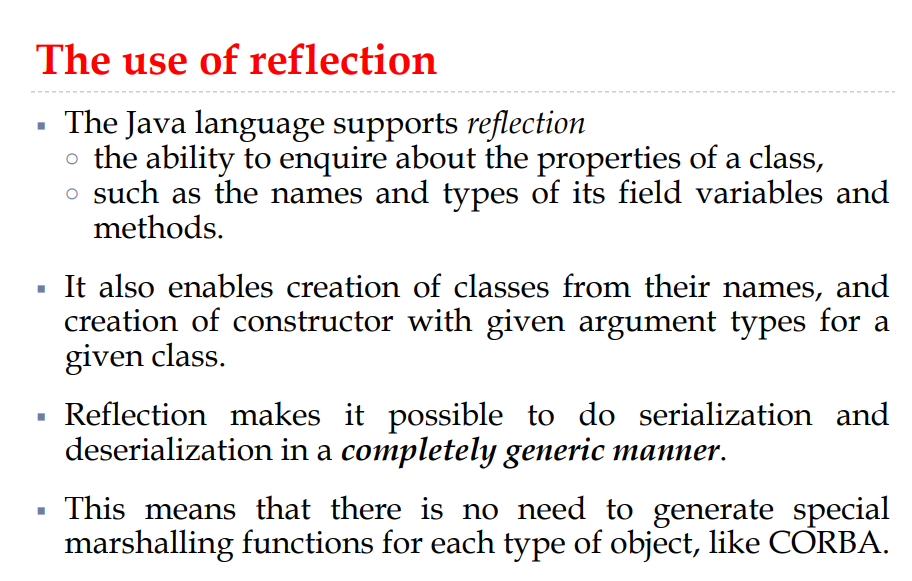
Therefore some information about the class of each object is included in serialized form to enable the recipient to load the appropriate class.



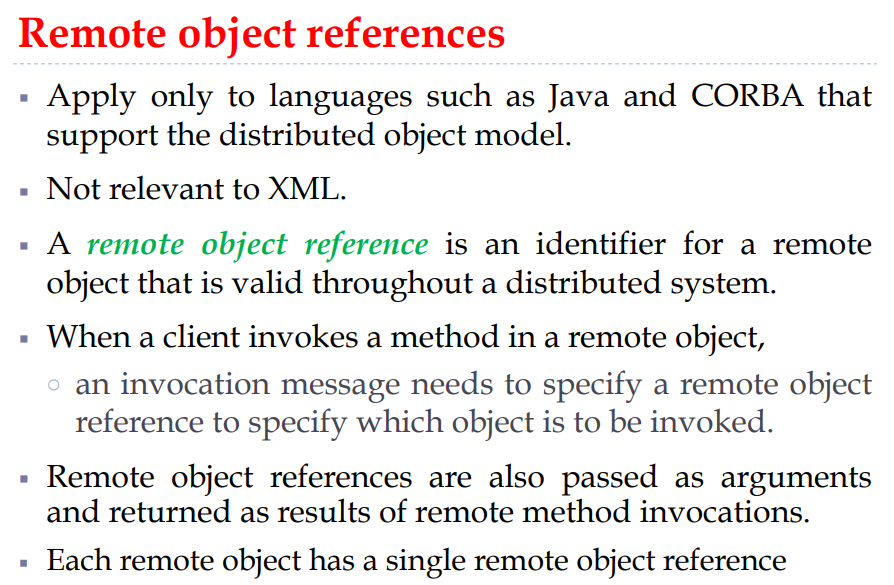


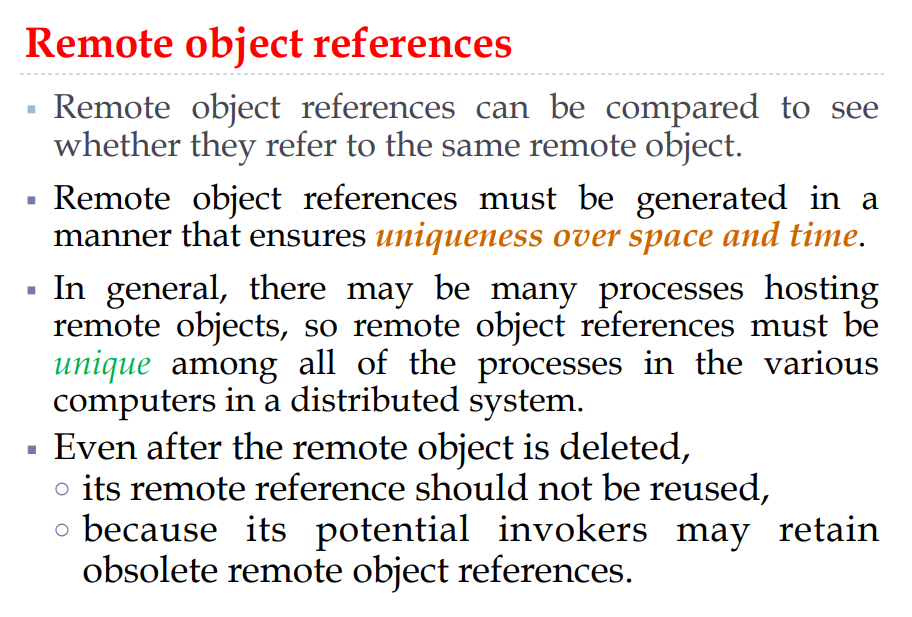


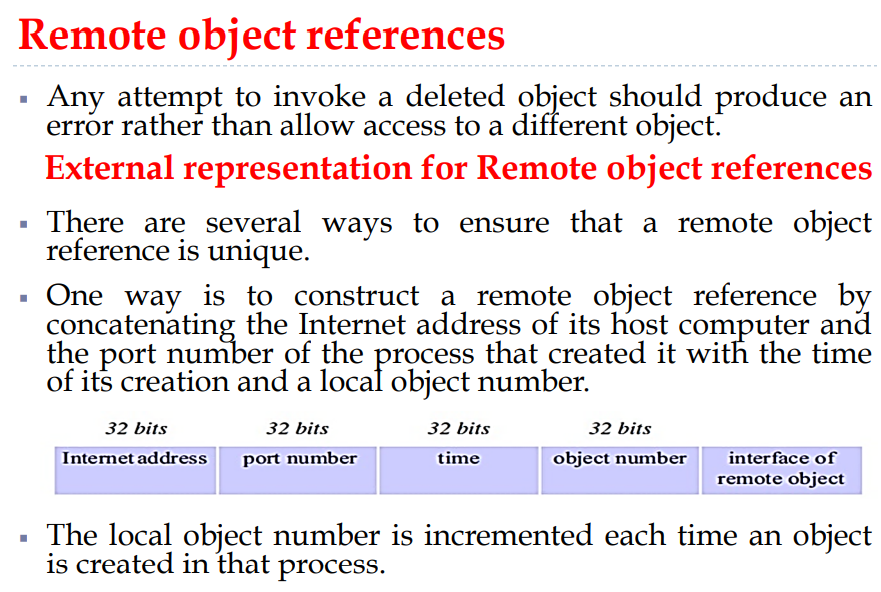


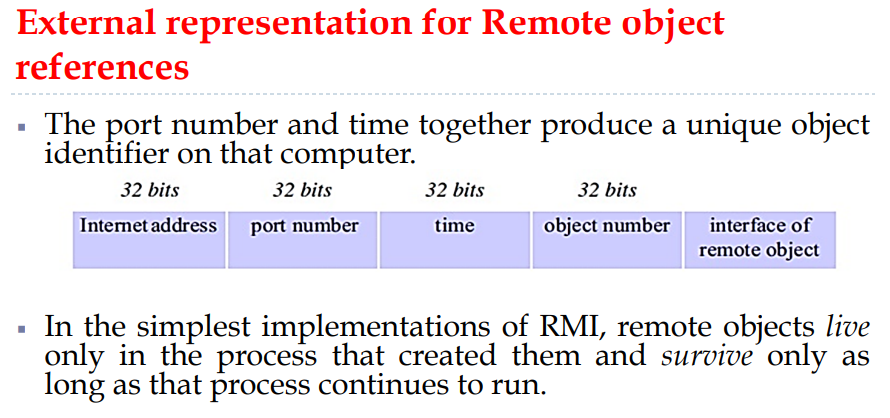


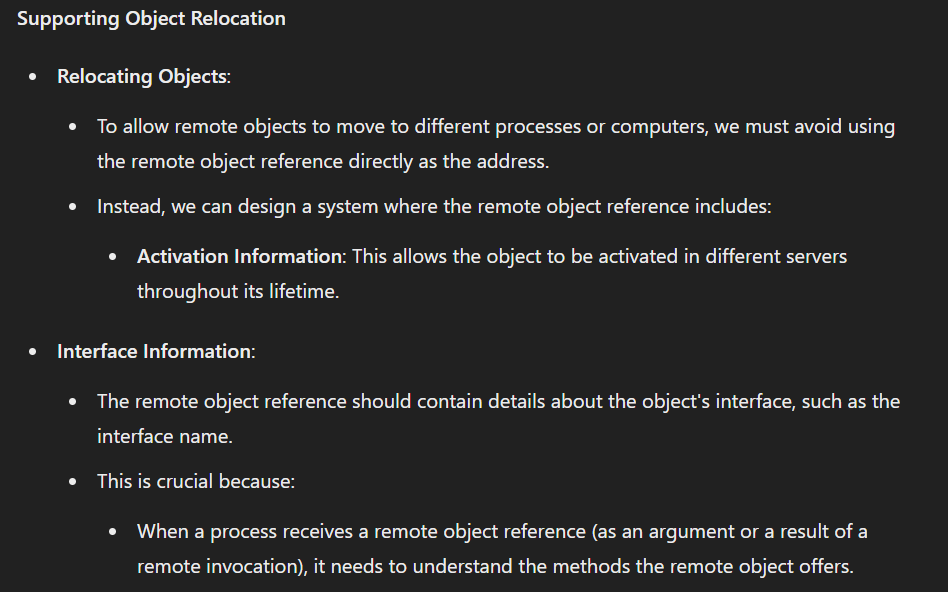








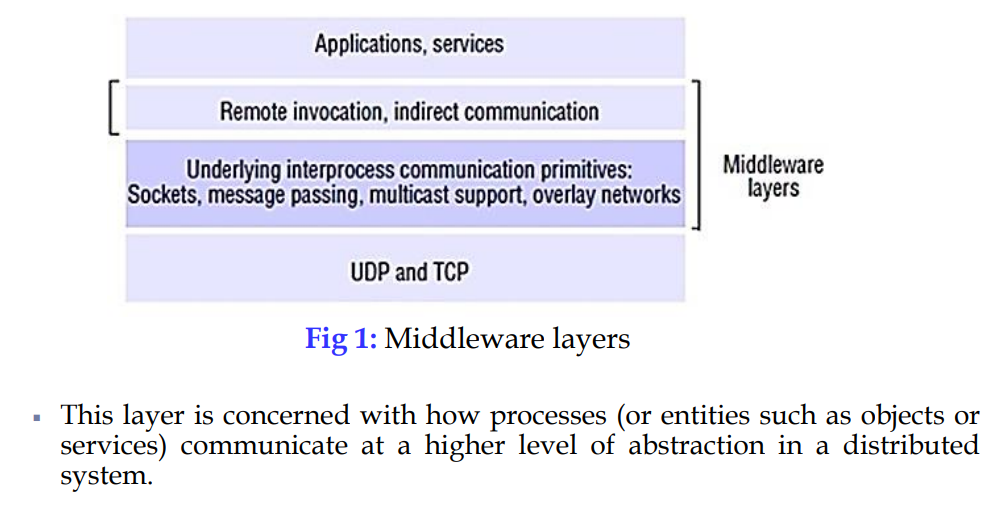




# **Lecture 14:**

## **Middleware Layers**

The middleware layers are essential components that facilitate communication and interaction between different software applications, particularly in a distributed computing environment. Here’s a detailed explanation of the middleware layers involved in remote method invocation (RMI):



### **Steps Involving Middleware Layers in Remote Method Invocation**

1. **Client Side**:

The client makes a method call on the client stub.

The client stub marshalls the method parameters into a byte stream.

The marshalled data is sent over the network to the server.

1. **Server Side**:

The server skeleton receives the marshalled data and unmarshalls it to extract the method parameters.

The skeleton invokes the appropriate method on the actual remote object.

After the method execution, the skeleton marshalls the result into a byte stream.

The marshalled result is sent back to the client over the network.

1. **Client Side (Continued)**:

The client stub receives the marshalled result and unmarshalls it to obtain the method’s return value.

The client stub returns the result to the client application.

### **Importance of Middleware Layers**

1. **Abstraction**: Middleware layers abstract the complexities of network communication, allowing developers to focus on the business logic rather than the intricacies of data transmission.
2. **Interoperability**: Middleware facilitates communication between heterogeneous systems, enabling applications written in different languages and running on different platforms to interact seamlessly.
3. **Scalability**: Middleware supports the development of scalable distributed applications by managing the underlying communication and data exchange efficiently.
4. **Reliability**: Middleware layers implement error handling, data retransmission, and other mechanisms to ensure reliable communication between distributed components.
5. **Security**: Middleware can include security features such as data encryption, authentication, and authorization to protect the integrity and confidentiality of the communication.

# **Lecture 17:**

## **Similarities Between RPC and RMI**

| **Feature** | **Description** |
| --- | --- |
| **Remote Communication** | Both enable programs to communicate and execute code on remote systems. |
| **Client-Server Model** | Both follow a client-server model where a client makes a request to a server, which processes it and returns a response. |
| **Use of Stubs** | Both use stub and skeleton mechanisms to handle remote communication transparently. |
| **Request-Reply Protocol** | Both typically use a request-reply protocol to send and receive messages. |

## **Differences Between RPC and RMI**

| **Feature** | **RPC (Remote Procedure Call)** | **RMI (Remote Method Invocation)** |
| --- | --- | --- |
| **Paradigm** | Procedure-oriented (function calls) | Object-oriented (method calls on objects) |
| **Programming Language** | Language-agnostic; can be used in various programming languages | Typically tied to Java, as it involves invoking methods on Java objects |
| **Interface Definition** | Uses an Interface Definition Language (IDL) to define the interface | Uses Java interfaces to define remote methods |
| **Data Representation** | Marshals and unmarshals data into a format suitable for transmission | Uses Java serialization to marshal and unmarshal objects |
| **Exception Handling** | Requires explicit handling of remote exceptions | Remote exceptions are part of the method signature and must be handled |
| **Type Safety** | Depends on the implementation; may not be type-safe | Type-safe, as it uses Java's strong typing system |
| **Object Passing** | Limited to passing data structures (structs, arrays, etc.) | Can pass objects by reference or by value (using serialization) |
| **Security** | Basic security mechanisms; depends on the protocol used (e.g., SSL/TLS) | More advanced security features (e.g., Java Security Manager, SSL/TLS) |
| **Transparency** | Aims to make remote calls look like local calls but usually requires some manual handling | Aims for full transparency, making remote method calls look like local calls |
| **Performance** | Generally lightweight and can be faster due to simpler data structures | Can be slower due to object serialization/deserialization overhead |
| **State Management** | Stateless or limited state management | Supports stateful objects and persistent state |
| **Service Registration** | Uses service discovery mechanisms like directory services | Uses Java Naming and Directory Interface (JNDI) for registering and locating services |
| **Implementation Complexity** | Typically simpler due to procedure-oriented design | More complex due to object-oriented design and additional features |

## **Design Issues in RMI:**

First three issues same as RPC, Programming with Interfaces, Call Semantics, Level of Transparency  
  
**Client-Server Architecture**

* **Client-Server Model**: In RMI, a client sends a request to a server to call a method on an object. The server executes the method and sends the result back to the client.
* **Chains of Invocations**: Sometimes, objects in one server might need to call methods on objects in another server, creating a chain of requests.

**Transition to Distributed Objects**

* **What it Means**: Handling objects that are spread across different computers.
* **Example**: Suppose your calculator service is distributed across multiple servers. You need to manage these objects and their locations.

**Concurrent RMIs from Objects**

When using RMI (Remote Method Invocation), different clients from various computers might try to call methods on the same remote object at the same time. This concurrent access can lead to conflicts and inconsistent data if not managed properly.

## **Distributed object model:**

The distributed object model is a framework used to enable communication and interaction between objects across different machines in a network, allowing them to work together as if they were on the same machine.

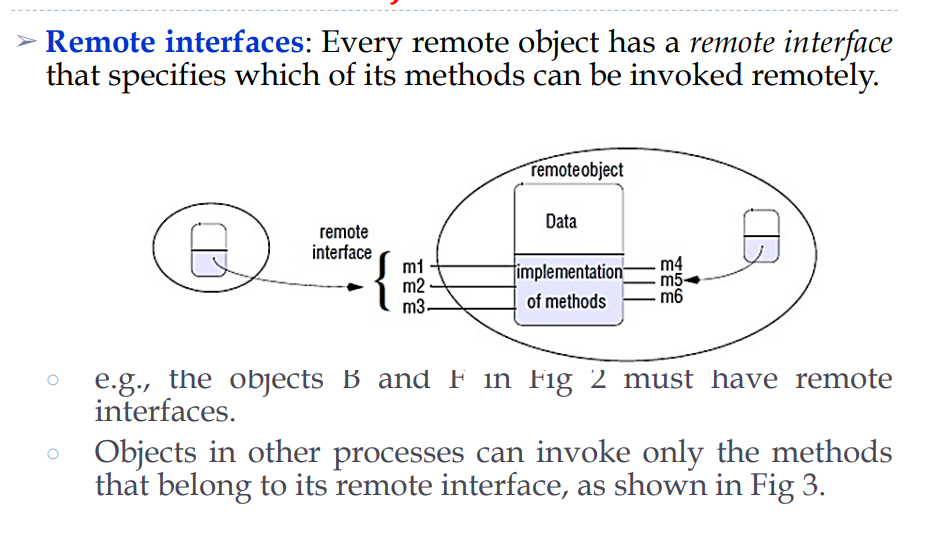
Is k bad explain krdenge:

Object

MEssage

Remote object references:

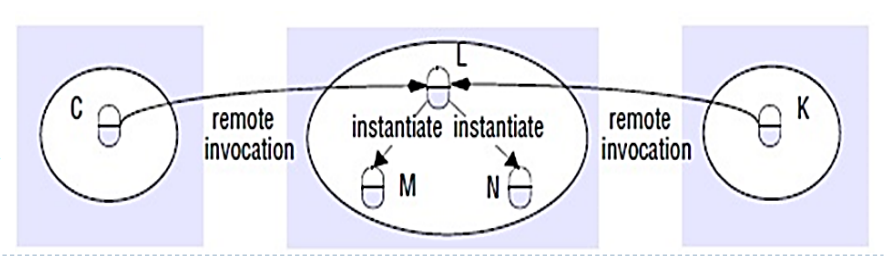
Remote interfaces:



**Actions** start with method invocations and can lead to further invocations or the creation of new objects.

**Object Instantiation**:

* **Local Instantiation**: When an action involves creating a new object, that object usually resides within the same process where the instantiation request was made. For instance, if a method call on object A triggers the creation of a new object, the new object typically lives in the process where object A's method was executed.
* **Remote Instantiation**: To facilitate remote object instantiation, distributed systems often provide mechanisms to allow objects to instantiate other objects remotely. For example, if object L has a method for creating new objects, invoking this method remotely (from objects C or K) can result in the instantiation of objects M and N on different machines.

****

**Exceptions** can be related to distribution issues or method execution problems. Frameworks like RMI and CORBA provide mechanisms for handling these exceptions, including specific notation for application-level exceptions and standard mechanisms for distribution-related errors.

# **Lecture 20:**

## **Threads:**

A thread, also known as a lightweight process, is the smallest unit of processing that can be scheduled and executed by an operating system. Threads are a fundamental part of multithreading, where a single process can have multiple threads running concurrently, allowing for parallel execution of tasks within the same application.

**Key Characteristics of Threads**

**1. Shared Resources**:

* **Definition**: Threads within the same process share the same resources, such as memory, file handles, and other data.

**2. Independent Execution**:

* **Definition**: Each thread can execute independently of others, performing its own tasks and operations.

**3. Lightweight**:

* **Definition**: Threads are considered lightweight because they use fewer resources compared to processes. They have less overhead since they share the same memory space and resources within a process.

**Common Uses of Threads**

**1. User Interfaces (UI)**: To keep the user interface responsive while performing background tasks.

**2. Servers**: To handle multiple client requests at the same time.

**3. Real-time Systems**: To execute multiple real-time tasks in parallel, ensuring timely and predictable responses.

**4. Simulations**: To run different parts of a simulation simultaneously, improving efficiency and accuracy.

## **How Threads are Used in RPC Systems**

**1. Concurrent Request Handling**:

* **Purpose**: To handle multiple remote procedure calls (RPCs) at the same time.
* **Explanation**: In an RPC system, each incoming request can be processed by a separate thread. This allows the system to manage multiple client requests simultaneously without having to wait for one request to complete before starting another.
* **Example**: A web service that handles multiple API requests from different clients concurrently by assigning each request to a different thread.

**2. Improved Responsiveness**:

* **Purpose**: To ensure that the system remains responsive while processing RPCs.
* **Explanation**: By using threads, the RPC server can continue to accept and process new requests even if some threads are busy with other operations. This prevents the server from becoming unresponsive or slow due to high load.
* **Example**: A file server that uses threads to handle read and write requests, ensuring that the server remains responsive to new requests even when other threads are busy processing existing ones.

**3. Resource Utilization**:

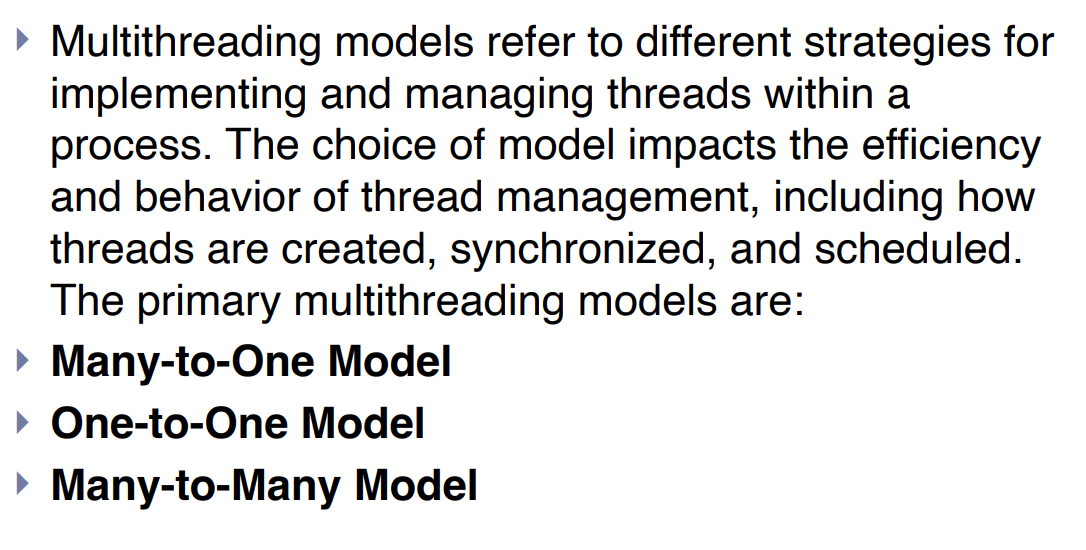
* **Purpose**: To efficiently use system resources by parallelizing work.
* **Explanation**: Threads allow an RPC system to make the best use of available CPU cores by performing multiple operations in parallel. This helps to maximize throughput and reduce idle time for resources.
* **Example**: In a database server handling multiple queries, threads can manage different queries concurrently, utilizing CPU cores effectively and improving overall performance.

## **Multithreading:**

# 

# 

## **Multithreading models:**



# 

### **1. Many-to-One Model**:

**Description**: In the many-to-one model, many user-level threads are mapped to a single kernel thread. Thread management is performed by the thread library in user space, which is efficient but has significant limitations.

**Advantages**:

* **Simple Management**: User-level threads are easy to create and manage.
* **Low Overhead**: Requires minimal kernel resources since only one kernel thread is needed.

**Disadvantages**:

* **No True Parallelism**: Only one thread can be executed at a time, regardless of the number of processors.
* **Blocking Issues**: If one user thread blocks, the entire process is blocked.
* **Limited Use of Multiprocessors**: Cannot take advantage of multiple processors for true parallel execution.

**Example**: Early versions of Java used the many-to-one model for threading.

### **2. One-to-One Model**:

**Description**:In the one-to-one model, each user-level thread maps to a separate kernel thread. This model provides more concurrency than the many-to-one model and is used by most modern operating systems.

**Advantages**:

* **True Parallelism**: Multiple threads can run in parallel on multiple processors.
* **Efficient Blocking Handling**: Blocking in one thread does not affect others since each thread has its own kernel thread.
* **Full Utilization of Multiprocessors**: Can fully utilize multiple processors for improved performance.

**Disadvantages**:

* **High Overhead**: Creating a large number of threads can lead to significant overhead due to each thread having a corresponding kernel thread.
* **Resource Intensive**: Requires more resources (memory and CPU) to manage each kernel thread.

**Example**: Windows and Linux operating systems use the one-to-one model for threads.

### **3. Many-to-Many Model**:

* **Description**: In the many-to-many model, many user-level threads are mapped to a smaller or equal number of kernel threads. The model allows the operating system to create sufficient kernel threads to handle multiple user threads efficiently.

**Advantages**:

* **Balanced Resource Utilization**: Provides a balanced use of resources, reducing overhead while still allowing multiple threads to be scheduled for execution.
* **Concurrency and Parallelism**: Allows concurrency and, depending on the number of kernel threads, some degree of parallelism.
* **Flexible Management**: Can handle multiple threads without the high overhead of one-to-one mapping.

**Disadvantages**:

* **Complex Implementation**: More complex to implement and manage than many-to-one or one-to-one models.
* **Performance Overhead:** Performance overhead due to the need to manage the mapping between user-level and kernel-level threads.

**Example**: Some implementations of Solaris use the many-to-many model

## **Support for Multithreading**

Support for multithreading can be provided at both the user level and the kernel level. Each level has its own mechanisms for creating, managing, and synchronizing threads, with different advantages and trade-offs.

## **User-Level Threads:**

User-level threads are managed by a user-level library or runtime, not the operating system kernel. All thread operations, such as creation, scheduling, and synchronization, are performed in user space.

**Advantages**:

* **Efficiency**: User-level thread operations are fast because they do not involve system calls, which can be slow.
* **Portability**: User-level threading libraries can be implemented on any operating system, as they do not rely on kernel support.
* **Customization**: Developers have fine control over the scheduling and management policies of threads.

**Disadvantages**:

* **Blocking System Calls**: If a thread makes a blocking system call, the entire process is blocked because the kernel is unaware of the user-level threads.
* **No True Parallelism**: On multiprocessor systems, user-level threads cannot achieve true parallelism because the kernel sees only one thread per process.

.

# 

# 

# 

# **Before mids:**

## **Amdahl's Law**

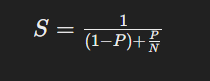
Amdahl's Law is a formula used in computer science to find the maximum improvement in speed or performance of a system when only part of the system is improved. It is particularly used to analyze the potential benefits of parallel computing.

### **Definition**

Amdahl's Law states that the overall performance improvement gained by optimizing a part of a system is limited by the fraction of the system that is not improved. In simpler terms, it tells us that the speedup of a program from parallelization is limited by the sequential portion of the program.

### **Formula**

Amdahl's Law is mathematically expressed as:



Where:

* S is the overall speedup of the system.
* P is the proportion of the program that can be parallelized.
* 1−P is the proportion of the program that is sequential and cannot be parallelized.
* N is the number of processors.

## **Berkeley Algorithm**

The Berkeley Algorithm, also known as the Berkeley Clock Synchronization Algorithm, is used to synchronize the clocks of computers in a network. This algorithm is particularly useful in distributed systems where accurate and synchronized time is crucial for coordinating events and actions.

### **Key Concepts**

1. **Master-Slave Architecture**:
   * A master node (often called the time server) is responsible for synchronizing the clocks of the other nodes (slave nodes) in the network.
   * The master node periodically checks the time of each slave node and calculates the average time to adjust the clocks accordingly.
2. **Clock Synchronization Process**:
   * The master node sends a request to all slave nodes asking for their current time.
   * Each slave node responds with its current time.
   * The master node collects all the time responses and calculates an average time (excluding any outliers if necessary).
   * The master node then sends an adjustment command to each slave node, specifying how much each node should adjust its clock to align with the average time.

## 

## **Memory Architectures:**

## **Shared Memory:**

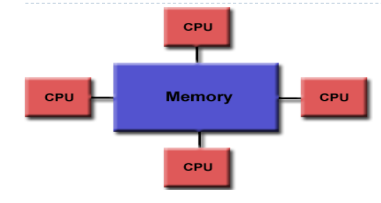
* In a shared memory system, multiple processors or CPUs have their own execution units but share the same physical memory space.
* Changes made by one CPU in the memory are immediately visible to all other processors.

**Advantages:**

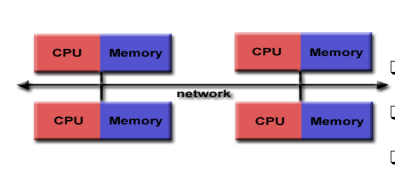
* Easy programming since all processors can directly access the same memory.
* Fast data sharing due to the proximity of memory to CPUs.

**Disadvantages:**

* Scalability issues arise when adding more CPUs, as it increases traffic on the shared memory-CPU path.
* Programmers must ensure proper synchronization and access to global memory.



## **Distributed Memory:**

****

In a distributed memory system, each processor has its own local memory, and changes made by one processor don't affect others. Communication between processors is needed to exchange data.

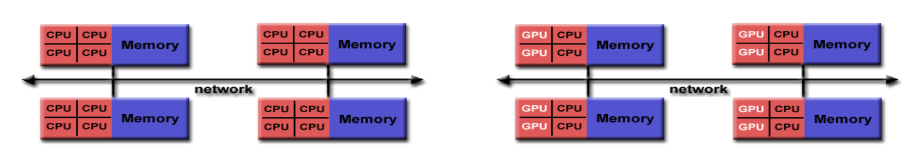
**Advantages:**

* Scalability, as memory scales with the number of CPUs.
* Each CPU can rapidly access its own memory without overhead related to global cache coherence.

**Disadvantages:**

* Programmers need to manage data communication between processors.
* Mapping existing data structures to this memory organization can be challenging.

## **Hybrid Distributed Shared Memory:**

****

* This architecture combines elements of both shared and distributed memory systems.
* Typically found in large-scale supercomputers where both shared memory machines and distributed memory machines are used.
* Each compute node has shared memory among its processors, but communication is needed between compute nodes to exchange data.

**Advantages and Disadvantages:**

* Inherits advantages and disadvantages from both shared and distributed memory architectures.
* Offers increased scalability but introduces increased programming complexity due to the need to manage both shared and distributed memory aspects.

## 

## **Principles of Vector Processing:**

1. **Simultaneous Processing of Data Elements:**

Principle: Vector processing involves the simultaneous execution of the same operation on multiple data elements. This is achieved through specialized vector instructions.

1. **Vector Registers:**

Principle: Vector processors use vector registers to store and manipulate multiple data elements. These registers allow efficient access to and processing of contiguous data.

1. **Vectorization of Code:**

Principle: Vector processing requires code to be written or compiled in a way that exploits the capabilities of vector instructions. Loops and operations are structured to take advantage of parallelism.

1. **Parallelism with a Single Instruction:**

Principle: Vector processors achieve parallelism by executing a single instruction on multiple data elements concurrently. This contrasts with scalar processors that operate on individual data items.

1. **Enhanced Throughput for Regular Data:**

Principle: Vector processing is particularly effective for regular and repetitive data structures, where the same operation is performed on a large set of data elements.

1. **Reduced Instruction Overhead:**

Principle: Vector processing minimizes instruction overhead by expressing operations on entire vectors with a single instruction, reducing the need for individual instructions for each data element.

1. **Efficient Memory Access:**

Principle: Vector processors often implement techniques like vector prefetching and caching to optimize memory access patterns, ensuring efficient retrieval of vector data from memory.

## **Array processors**

Array processors are specialized computers made for handling arrays or matrices of data really efficiently. Here are the main things to know about them:

## **Characteristics of Array Processors:**

1. **Parallel Processing:** They're good at doing many things at once. Each part of the processor works on a different piece of the array at the same time.
2. **Specialized Instructions:** These processors come with special instructions just for working with arrays. These instructions help them do array operations really fast.
3. **Vector and Matrix Operations:** They're great at doing math on arrays and matrices, like adding, multiplying, or transforming them all at once.
4. **Memory Architecture:** The way they store and access data is designed to be super quick. They might have special memory setups to make sure they can get the data they need fast.
5. **High Throughput:** They're really good at handling big sets of data quickly, especially when the data follows a pattern. This makes them perfect for tasks like scientific simulations or big engineering projects.
6. **Scientific and Engineering Applications:** They're used a lot in science and engineering because those fields often deal with huge amounts of data that need to be crunched quickly. So, anything from simulating physical systems to processing signals can benefit from array processors.
7. **Data Parallelism:** Array processors are built around the idea of doing the same thing to lots of data at once. This fits perfectly with how arrays work, where you often want to apply the same operation to many elements simultaneously.

Array processors have their strengths and challenges. Let's break them down:

### **Advantages**:

1. **Efficiency in Parallel Operations:** They're really good at doing lots of things at once with arrays, which makes them super fast for computations.
2. **Optimized for Mathematical Operations:** These processors are designed specifically for doing math, which is great for scientific and engineering tasks that involve lots of calculations.
3. **High Throughput:** Because they can work on many parts of the data simultaneously, they can handle big datasets really quickly.

### Challenges:

1. **Limited Applicability:** They're specialized machines, so they might not be the best choice for every kind of task. General-purpose processors might be better for some jobs.
2. **Programming Complexity:** Writing programs for array processors can be tricky because you have to explicitly tell the computer how to split up the work. This can make the programming more complex compared to regular programming.

## **SIMD**

SIMD stands for **Single Instruction, Multiple Data**. It is a parallel computing architecture that allows a single instruction to be executed simultaneously on multiple data points. SIMD is commonly used in applications that require the same operation to be performed on large sets of data, such as multimedia processing, scientific computing, and graphics rendering.

### **Applications**

1. **Multimedia Processing**:
   * Image and video processing often involve applying the same operation to many pixels simultaneously.
2. **Scientific Computing**:
   * Many scientific computations, such as matrix operations and simulations, benefit from SIMD by performing calculations on large datasets in parallel.
3. **Graphics Rendering**:
   * Graphics processing units (GPUs) use SIMD to render images by performing operations on multiple pixels or vertices simultaneously.
4. **Data Parallelism**:
   * Operations on large datasets, such as in machine learning and big data analytics, can leverage SIMD for efficient computation.

### **Examples**

1. **Vector Processing**
2. **Matrix Multiplication**

**Benefits**

* **Increased Throughput**: By performing the same operation on multiple data points simultaneously, SIMD can significantly increase processing speed.
* **Efficiency**: Reduces the instruction overhead since one instruction controls multiple data operations.
* **Cost-Effective**: Utilizes the hardware efficiently by keeping multiple processing units busy.

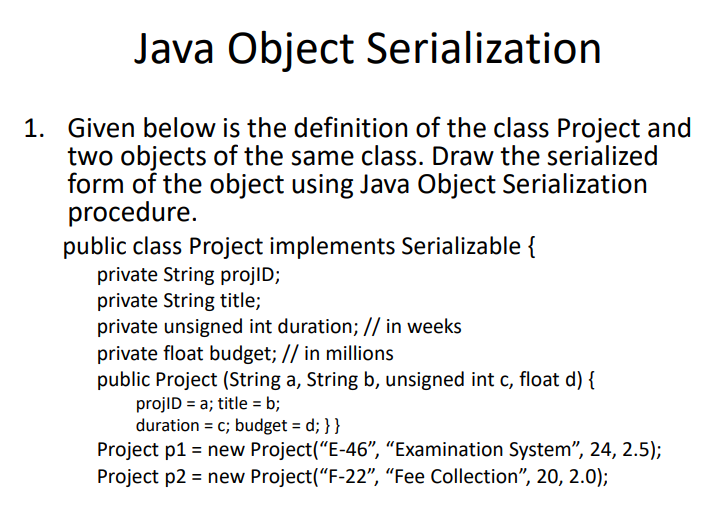
### **Limitations**

* **Uniformity Requirement**: SIMD is most effective when the same operation is applied to many data points. It is less useful for operations that require different instructions for different data.
* **Data Alignment**: Data needs to be properly aligned in memory, which can sometimes be complex to manage.

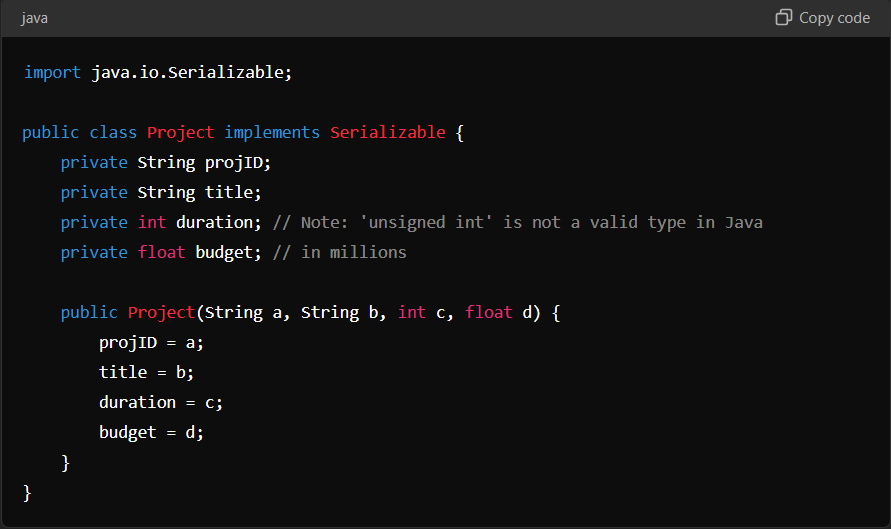
# 

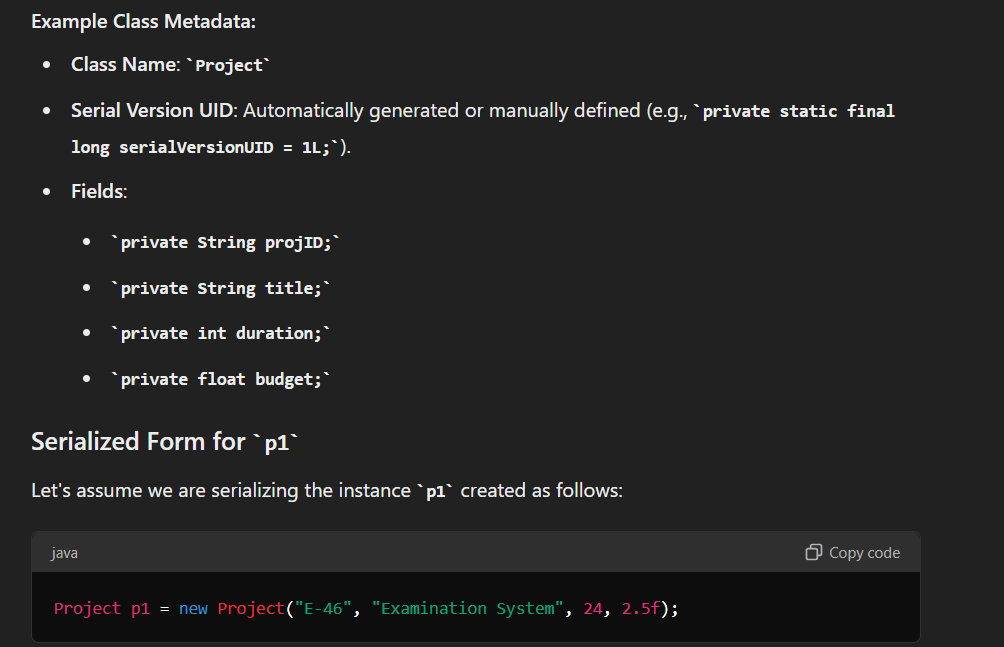
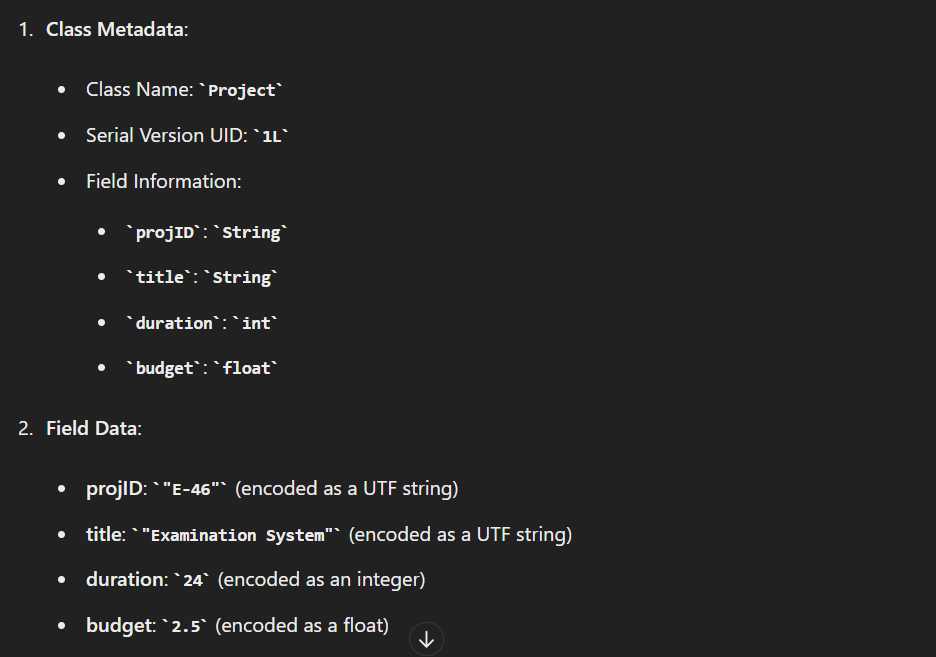
# 

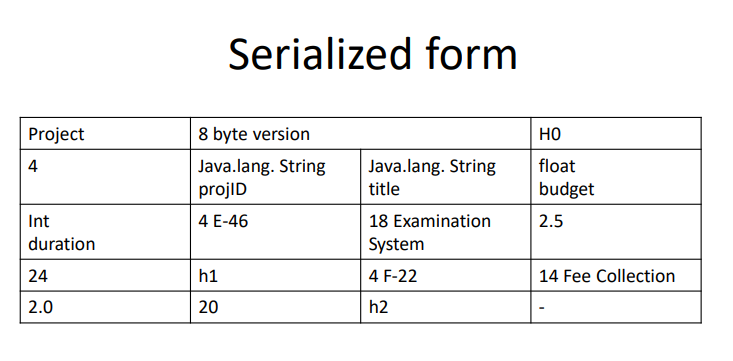
# **Numericals:**

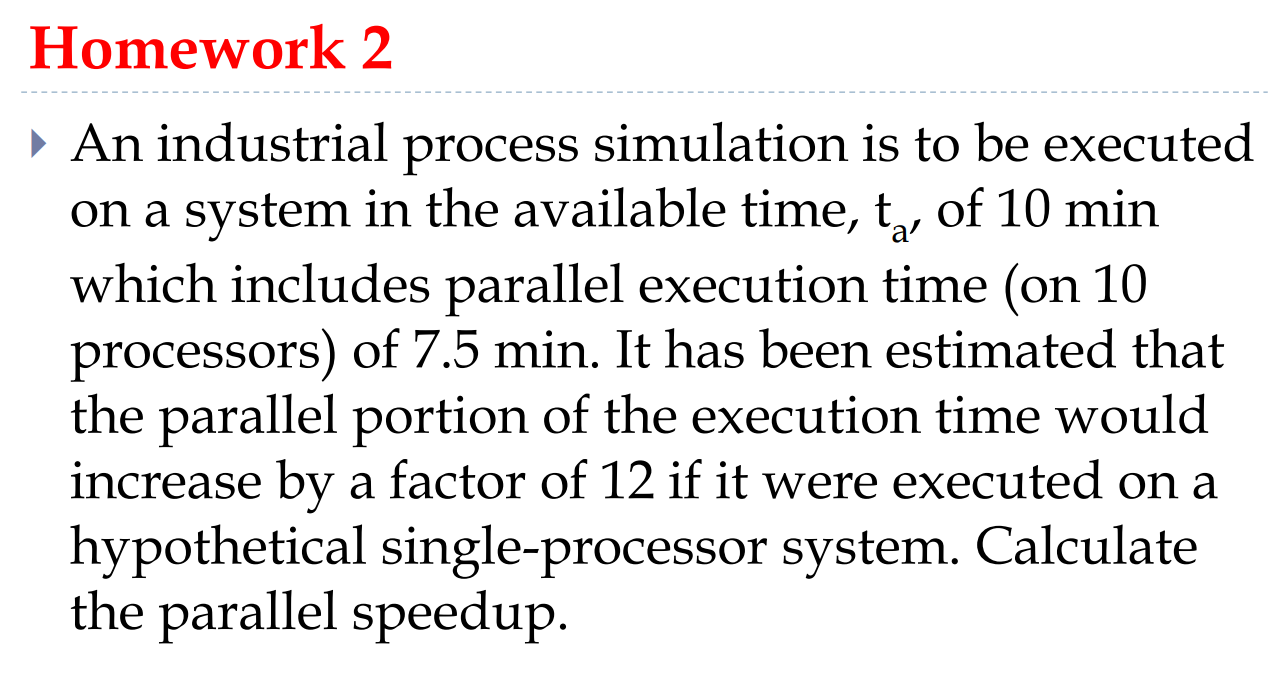


## **Solution:**









To calculate the parallel speedup of the industrial process simulation, we need to follow these steps:

1. **Identify the given information**:
   * Total available time (tat\_ata​): 10 minutes.
   * Parallel execution time on 10 processors (tpt\_ptp​): 7.5 minutes.
   * The parallel portion of the execution time would increase by a factor of 12 if executed on a single processor.
2. **Determine the sequential and parallel components of the execution time**:
   * Let tst\_sts​ be the sequential (non-parallelizable) part of the execution time.
   * The total execution time tat\_ata​ is the sum of the sequential and parallel parts: ta=ts+tpt\_a = t\_s + t\_pta​=ts​+tp​
3. **Calculate the sequential part of the execution time**:
   * Given ta=10t\_a = 10ta​=10 minutes and tp=7.5t\_p = 7.5tp​=7.5 minutes, the sequential time tst\_sts​ is: ts=ta−tp=10−7.5=2.5 minutest\_s = t\_a - t\_p = 10 - 7.5 = 2.5 \text{ minutes}ts​=ta​−tp​=10−7.5=2.5 minutes
4. **Calculate the parallel portion on a single processor**:
   * The parallel portion of the execution time on a single processor is 12 times the parallel execution time on 10 processors: tp1=12×tp=12×7.5=90 minutest\_{p1} = 12 \times t\_p = 12 \times 7.5 = 90 \text{ minutes}tp1​=12×tp​=12×7.5=90 minutes
5. **Calculate the total execution time on a single processor**:
   * The total execution time on a single processor (tsinglet\_{single}tsingle​) includes the sequential part plus the parallel portion on a single processor: tsingle=ts+tp1=2.5+90=92.5 minutest\_{single} = t\_s + t\_{p1} = 2.5 + 90 = 92.5 \text{ minutes}tsingle​=ts​+tp1​=2.5+90=92.5 minutes
6. **Calculate the parallel speedup**:
   * Parallel speedup (SSS) is the ratio of the execution time on a single processor to the execution time on multiple processors: S=tsingleta=92.510=9.25S = \frac{t\_{single}}{t\_a} = \frac{92.5}{10} = 9.25S=ta​tsingle​​=1092.5​=9.25

### **Explanation:**

* **Total Available Time (tat\_ata​)**: The total time allocated for the simulation is 10 minutes.
* **Parallel Execution Time (tpt\_ptp​)**: The time taken by the parallel portion of the simulation when executed on 10 processors is 7.5 minutes.
* **Sequential Time (tst\_sts​)**: The time for the part of the simulation that cannot be parallelized is calculated as 2.5 minutes.
* **Single-Processor Parallel Time (tp1t\_{p1}tp1​)**: The parallel portion's execution time increases by a factor of 12 when executed on a single processor, resulting in 90 minutes.
* **Total Single-Processor Time (tsinglet\_{single}tsingle​)**: The total time required to run the simulation on a single processor is 92.5 minutes.
* **Parallel Speedup (SSS)**: The improvement in execution time due to parallelization, calculated as the ratio of single-processor time to multi-processor time, is 9.25.

Therefore, the parallel speedup of the industrial process simulation is 9.259.259.25. This means the simulation runs 9.25 times faster on the 10-processor system compared to a single-processor system.

