

# CS 3551 DISTRIBUTED COMPUTING

## UNIT I INTRODUCTION

8

Introduction: Definition-Relation to Computer System Components – Motivation – Message -  
Passing Systems versus Shared Memory Systems – Primitives for Distributed Communication –  
Synchronous versus Asynchronous Executions – Design Issues and Challenges; A Model of  
Distributed Computations: A Distributed Program – A Model of Distributed Executions – Models of  
Communication Networks – Global State of a Distributed System.

LIKE



COMMENT



SHARE



SUBSCRIBE



# Primitives for Distributed Computing



| Function           | Parameters  | Need                                  |
|--------------------|---|---------------------------------------|
| ① Send()           | <ol style="list-style-type: none"><li>1. destination, ✓</li><li>2. <u>buffer in the user space, containing the data to be sent.</u></li></ol>                                       | To send data from a system to another |
| ② <u>Receive()</u> | <ol style="list-style-type: none"><li>1. <u>source from which the data is to be received</u></li><li>2. <u>and the user buffer into which the data is to be received.</u></li></ol> | To receive data from another system   |

| Sl | Primitive    | Shortcut                          |
|----|--------------|-----------------------------------|
| 1  | Synchronous  | I care for you, you care for me   |
| 2  | Asynchronous | I don't care for you              |
| 3  | Blocking     | I care for my task completion     |
| 4  | Non-blocking | I don't even care for my own task |

# Let's Understand With Example

Alice → Bob

Send

receive

## 1. Synchronous Primitives:

- Imagine Alice wants to send a message to Bob, and they both have to make sure the message is successfully delivered and received.
- Alice writes her message on a piece of paper and hands it to Bob.
- Alice doesn't consider her task complete until she sees Bob read and understand the message.
- Bob doesn't consider his task complete until he has read and understood Alice's message.
- So, in this synchronous scenario, both Alice and Bob wait for each other to complete their parts of the process before they can move on to other tasks.

## 2. **Asynchronous Primitives:**

- In this case, Alice wants to send a message to Bob, but she doesn't need to wait for Bob to read it before doing something else.
- Alice writes her message on a piece of paper and hands it to Bob.
- After handing over the message, Alice doesn't wait for Bob to read it. She can go about her other tasks immediately.
- Bob receives the message and reads it whenever he has time, but Alice doesn't have to be present while he does so.

Send()

### 3. Blocking Primitives:

- In both synchronous and asynchronous scenarios, if Alice and Bob use blocking primitives, it means they can't do anything else until their message exchange is complete.
- So, if they use blocking primitives and the message is synchronous, both Alice and Bob are "blocked" from other activities until the message is delivered and read.
- Similarly, if they use blocking primitives with asynchronous messaging, Alice still can't do anything else until the message is sent, even though Bob may read it at his convenience.

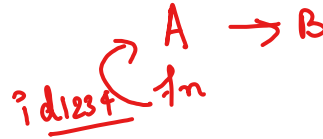
#### 4. **Non-blocking Primitives:**

(Initialise)

User buffer  $\rightarrow$  kernel  $\Rightarrow$  n/w  $\rightarrow$

- In this scenario, Alice and Bob use non-blocking primitives.
- If Alice sends Bob a message non-blockingly, she can continue with her tasks immediately after handing him the message, without waiting for him to read it.
- If Bob receives the message non-blockingly, he can continue with his tasks immediately after receiving the message, even if he hasn't read it yet.

# Need of Handles $\Rightarrow id$



1. **Non-blocking Primitives**: These are asynchronous operations that don't make your program wait for them to finish.
2. **Return Parameter (Handle)**: When you use a non-blocking primitive, you get a handle, which is like an ID for that operation.
3. **Checking Completion**:
  - a. **Polling**: You can repeatedly check the handle's status to see if the operation is done.
  - b. **Waiting with Handles**: You can use a Wait function with handles to pause your program until at least one operation is finished.
4. **Blocking Wait**: If your program depends on the non-blocking operation, you can use a blocking Wait to stop everything until that operation is complete.



# Key Points

- **Synchronous primitives** A *Send* or a *Receive* primitive is *synchronous* if both the *Send()* and *Receive()* handshake with each other. The processing for the *Send* primitive completes only after the invoking processor learns that the other corresponding *Receive* primitive has also been invoked and that the receive operation has been completed. The processing for the *Receive* primitive completes when the data to be received is copied into the receiver's user buffer.
- **Asynchronous primitives** A *Send* primitive is said to be *asynchronous* if control returns back to the invoking process after the data item to be sent has been copied out of the user-specified buffer.  
It does not make sense to define asynchronous *Receive* primitives.
- **Blocking primitives** A primitive is *blocking* if control returns to the invoking process after the processing for the primitive (whether in synchronous or asynchronous mode) completes.
- **Non-blocking primitives** A primitive is *non-blocking* if control returns back to the invoking process immediately after invocation, even though the operation has not completed. For a non-blocking *Send*, control returns to the process even before the data is copied out of the user buffer. For a non-blocking *Receive*, control returns to the process even before the data may have arrived from the sender.





LIKE



COMMENT

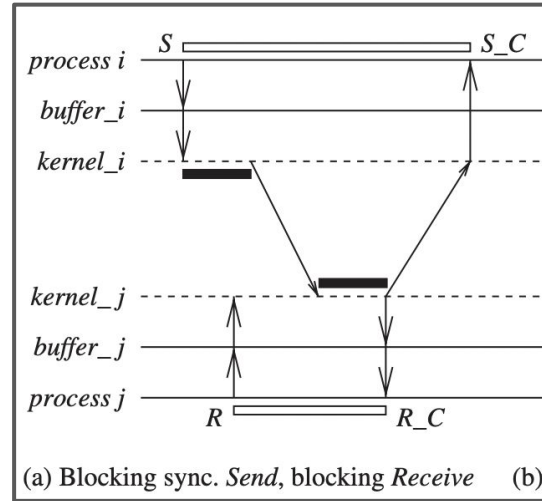
SHARE



SUBSCRIBE

User bu → kernel bu → net → receiver

# Primitives Part 2a





## Send

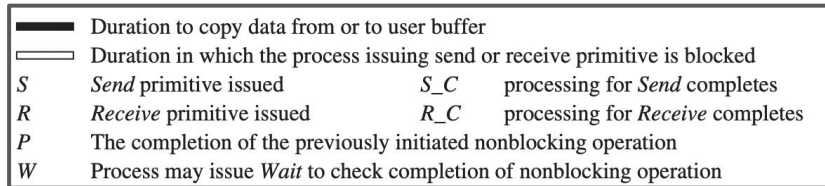
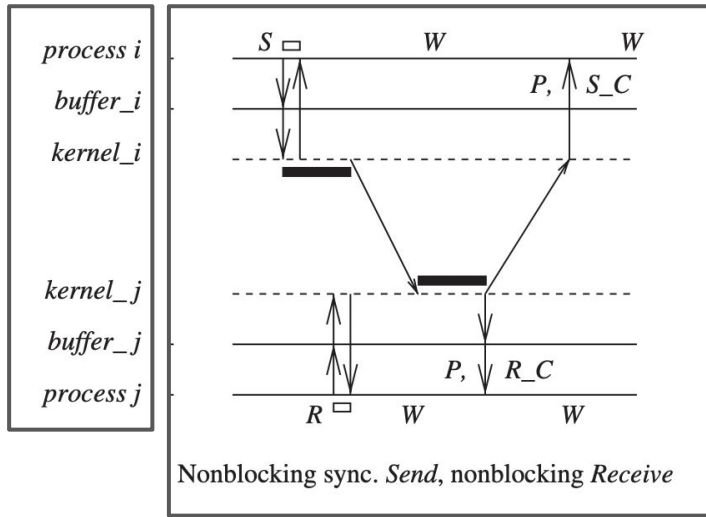
- The data gets copied from the user buffer to the kernel buffer and is then sent over the network.
- After the data is copied to the receiver's system buffer and a Receive call has been issued, an acknowledgement is sent to sender.
- The process that invoked the Send operation and completes the Send.

## Receive

- The Receive call blocks until the data expected arrives and is written in the specified user buffer.
- Then control is returned to the user process

|   |  |
|---|--|
|  | Duration to copy data from or to user buffer                               |
|  | Duration in which the process issuing send or receive primitive is blocked |
| <i>S</i>  | <i>Send</i> primitive issued   |
| <i>S_C</i>  | processing for <i>Send</i> completes                                       |
| <i>R</i>  | <i>Receive</i> primitive issued  |
| <i>R_C</i>  | processing for <i>Receive</i> completes                                    |
| <i>P</i>  | The completion of the previously initiated nonblocking operation           |
| <i>W</i>  | Process may issue <i>Wait</i> to check completion of nonblocking operation |

# Primitives Part 2b



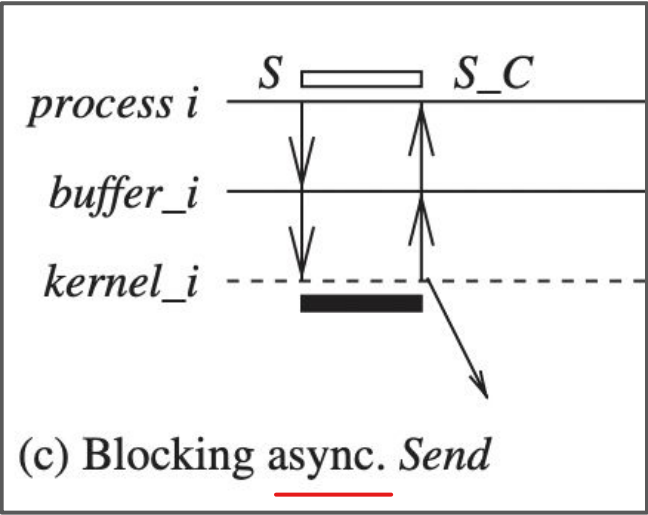
## Send

- Control returns back to the invoking process as soon as the copy of data from the user buffer to the kernel buffer is initiated
- Handle is returned as parameter which can be used to check the process completion.

## Receive



- When you make a *Receive* call, the system gives you a special ID (handle).
- Handle can be used to check if the non-blocking *Receive operation* is finished.
- The system sets this handle as "done" when the expected data arrives and is put in your specified place (buffer).

# Primitives Part 2c

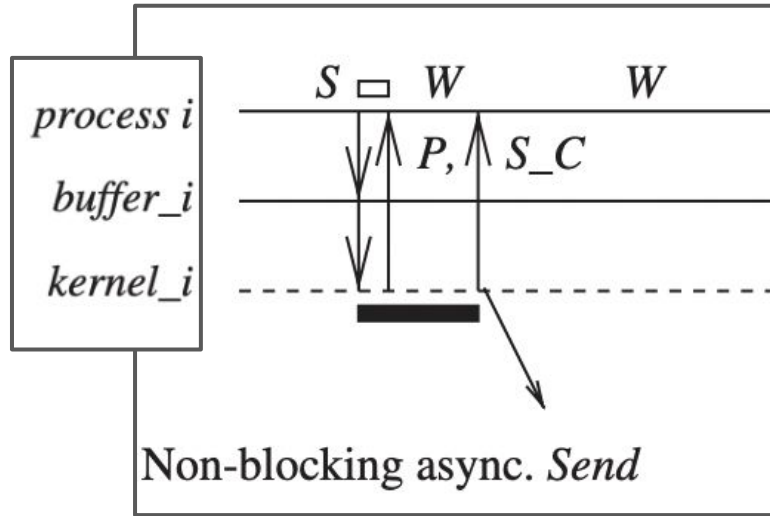


## Send

- The user process that invokes the Send is blocked until the data is copied from the user's buffer to the kernel buffer.



|   |  |
|---|--|
|  | Duration to copy data from or to user buffer                               |
|  | Duration in which the process issuing send or receive primitive is blocked |
| <i>S</i>  | <i>Send</i> primitive issued   |
| <i>S_C</i>  | processing for <i>Send</i> completes                                       |
| <i>R</i>  | <i>Receive</i> primitive issued  |
| <i>R_C</i>  | processing for <i>Receive</i> completes                                    |
| <i>P</i>  | The completion of the previously initiated nonblocking operation           |
| <i>W</i>  | Process may issue <i>Wait</i> to check completion of nonblocking operation |

# Primitives Part 2d



## Send

- The user process that invokes the Send is blocked until the transfer of the data from the user's buffer to the kernel buffer is initiated.
- Control returns to the user process as soon as this transfer is initiated, and handle is given back.
- The asynchronous Send completes when the data has been copied out of the user's buffer

|   |  |
|---|--|
|  | Duration to copy data from or to user buffer                               |
|  | Duration in which the process issuing send or receive primitive is blocked |
| <i>S</i>  | <i>Send</i> primitive issued   |
| <i>S_C</i>  | processing for <i>Send</i> completes                                       |
| <i>R</i>  | <i>Receive</i> primitive issued  |
| <i>R_C</i>  | processing for <i>Receive</i> completes                                    |
| <i>P</i>  | The completion of the previously initiated nonblocking operation           |
| <i>W</i>  | Process may issue <i>Wait</i> to check completion of nonblocking operation |

# Key Points

1. A synchronous Send is easier to use from a programmer's perspective because the handshake between the Send and the Receive makes the communication appear instantaneous, thereby simplifying the program logic.
2. The non-blocking asynchronous Send (see Figure 1.8(d)) is useful when a large data item is being sent because it allows the process to perform other instructions in parallel with the completion of the Send.
3. The non-blocking synchronous Send (see Figure 1.8(b)) also avoids the potentially large delays for handshaking, particularly when the receiver has not yet issued the Receive call.
4. The non-blocking Receive (see Figure 1.8(b)) is useful when a large data item is being received and/or when the sender has not yet issued the Send call, because it allows the process to perform other instructions in parallel with the completion of the Receive.



# Process Synchrony



dance  
(10)

2min

1. **Processor Synchrony:** This means that all the computers or processors in a system work together perfectly, like synchronized dancers following the same rhythm. Their internal clocks are all perfectly in sync.
2. **In Distributed Systems:** In reality, achieving perfect synchrony among all processors in a distributed system is very difficult or impossible. So, what we do instead is synchronize them in a different way.
3. **Synchronization at a Higher Level:** Instead of making every little action perfectly synchronized, we group many actions into larger chunks called "steps." Think of these steps like dance routines.
4. **Barrier Synchronization:** To make sure these steps are performed in sync, we use a mechanism called "barrier synchronization." It's like a checkpoint in a dance routine. No dancer can move to the next step until everyone has completed the current one. Similarly, in a distributed system, no processor can move on to the next step of their work until all processors have finished their current step.

20sec  
(Sstep)  
Group

# Libraries and Standards



1. **MPI:** Message-Passing Interface
2. **PVM:** Parallel Virtual Machine
3. **RPC:** Remote Procedure Call
4. **DCE:** Distributed Computing Environment
5. **RMI:** Remote Method Invocation
6. **ROI:** Remote Object Invocation
7. **CORBA:** Common Object Request Broker Architecture
8. **DCOM:** Distributed Component Object Model

- In computer systems, there are many ways for programs to talk to each other, like sending messages or making remote calls. Different software products and scientific tools use their own special ways to do this.
- For example, some big companies use their custom methods, like IBM's CICS software. Scientists often use libraries called MPI or PVM. Commercial software often uses a method called RPC, which lets you call functions on a different computer like you would on your own computer.
- All these methods use something like a hidden network phone line (called "sockets") to make these remote calls work.
- There are many types of RPC, like Sun RPC and DCE RPC. There are also other ways to communicate, like "messaging" and "streaming."
- As software evolves, there are new methods like RMI and ROI for object-based programs, and big standardized systems like CORBA and DCOM.



LIKE



COMMENT

SHARE



SUBSCRIBE

# Synchronous vs Asynchronous Execution

---

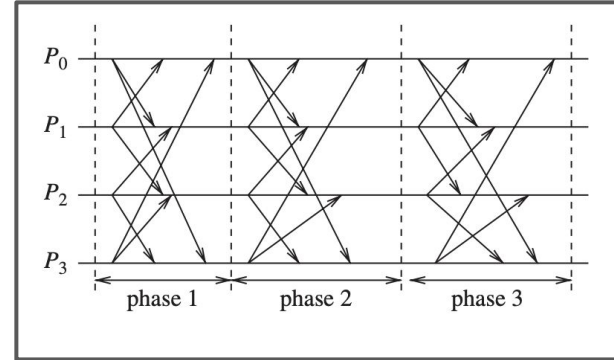
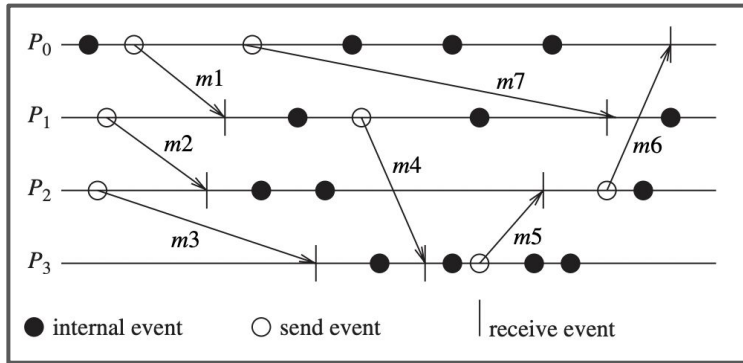
## UNIT I

## INTRODUCTION

8

Introduction: Definition-Relation to Computer System Components – Motivation – Message - Passing Systems versus Shared Memory Systems – Primitives for Distributed Communication – Synchronous versus Asynchronous Executions – Design Issues and Challenges; A Model of Distributed Computations: A Distributed Program – A Model of Distributed Executions – Models of Communication Networks – Global State of a Distributed System.

# Synchronous vs Asynchronous Execution



## Asynchronous

1. there is no processor synchrony and there is no bound on the drift rate of processor clocks,
2. message delays (transmission + propagation times) are finite but unbounded
3. there is no upper bound on the time taken by a process to execute a step

## Synchronous

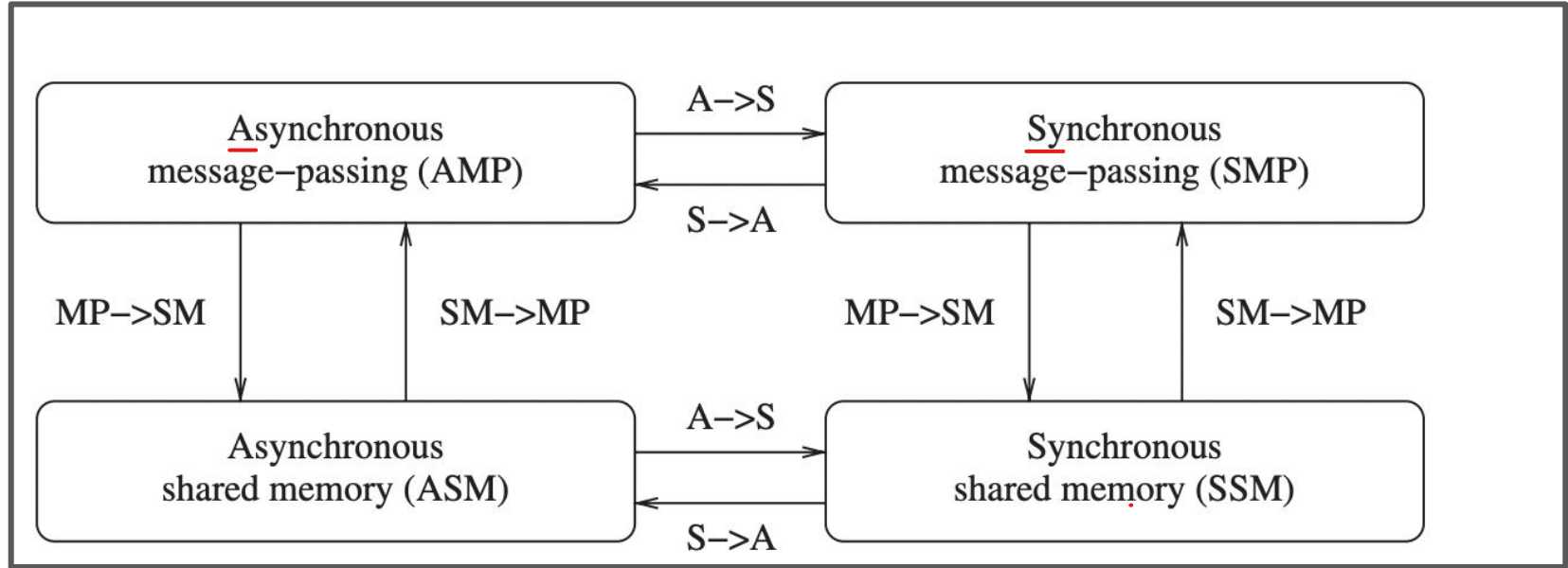
1. processors are synchronized and the clock drift rate between any two processors is bounded
2. message delivery (transmission + delivery) times are such that they occur in one logical step or round
3. there is a known upper bound on the time taken by a process to execute a step

# Key Points

- It is easier to design and verify algorithms assuming synchronous executions because of the coordinated nature of the executions at all the processes.
- It is practically difficult to build a completely synchronous system, and have the messages delivered within a bounded time. **(Simulated)**
- Thus, synchronous execution is an abstraction that needs to be provided to the programs. When implementing this abstraction, observe that the fewer the steps or “synchronizations” of the processors, the lower the delays and costs.



# Emulating Systems





LIKE



COMMENT



SHARE



SUBSCRIBE



# Design Issue and Challenges

## UNIT I





## INTRODUCTION

8

Introduction: Definition-Relation to Computer System Components – Motivation – Message - Passing Systems versus Shared Memory Systems – Primitives for Distributed Communication – Synchronous versus Asynchronous Executions – **Design Issues and Challenges;** A Model of Distributed Computations: A Distributed Program – A Model of Distributed Executions – Models of Communication Networks – Global State of a Distributed System.

1. From System Perspective ✓
  2. Algorithmic Challenges ✓
  3. Applications of distributed computing
-

# 1. From System Perspective ( CSCANS-PDF)

| Challenge   | Description  |
|---|--|
| Communication<br>            | designing appropriate mechanisms for communication among the processes in the network<br>Eg <u>RPC</u> , <u>ROI</u> , <u>Message oriented</u> vs <u>stream oriented</u> communication  |
| Synchronization<br>          | Mechanism for synch among processes.<br>Eg <u>Mutual exclusion</u> & <u>leader election</u> algos<br>Global <u>state recording</u> algo & <u>physical clocks</u> need synch  |
| Consistency and replication  | Replicate for performance but it is essential that replicated compute should be consistent.  |
| API and transparency<br>     | <ol style="list-style-type: none"><li>1. API Required for ease of use.</li><li>2. Access transparency hides differences in data representation on different systems and provides uniform operations to access system resources.</li><li>3. Location transparency makes the locations of resources transparent to the users.</li><li>4. Migration transparency allows relocating resources without changing names</li></ol> |

# 1. From System Perspective ( CSCANS-PDF)

| Challenge                                     | Description   |
|---|---|
| Naming  | Easy to use and robust <u>naming</u> for identifiers, and <u>addresses</u> is essential for locating resources and processes in a transparent and scalable manner.  |
| <u>Security &amp; Scalability</u>             | Involves various aspects of <u>cryptography</u> , <u>secure channels</u> , access control, key <u>management</u> – generation and distribution, authorization, and secure group <u>management</u><br>The system must be scalable to handle large loads.   |
| <u>Processes</u>                              | management of <u>processes</u> and threads at clients/servers; code migration; and the design of software and mobile agents.  |
| <u>Data storage and access</u>                | Schemes for <u>data storage</u> , and implicitly <u>for accessing</u> the data in a fast and scalable manner <u>across the network</u> are important for efficiency   |
| <u>Fault tolerance</u><br><br>1, 2, 3, 4<br>C | <ul style="list-style-type: none"><li>Fault tolerance requires maintaining correct and efficient operation in spite of any <u>failures</u> of links, nodes, and processes.</li><li>Process <u>resilience</u>, reliable <u>communication</u>, distributed commit, checkpointing and <u>recovery</u>, agreement and <u>consensus</u>, failure detection, and <u>self-stabilization</u> are some of the mechanisms to provide fault-tolerance.</li></ul> |

## 2. Algorithmic Challenges ( CREPT-MGW)

| Challenge  | Description  |
|--|--|
| group Communication, multicast, and ordered message delivery | <ol style="list-style-type: none"><li>1. A group is a <u>collection of processes</u> that share a <u>common context</u> and collaborate on a <u>common task within an application domain</u></li><li>2. Algorithms need to be designed to enable efficient group communication and group management <u>wherein</u> processes can join and leave groups dynamically</li><li>3. Specify order of <u>delivery when multiple process send message concurrently</u></li></ol> |
| Replication and consistency ✓                                | <ul style="list-style-type: none"><li>• Replicate for performance.</li><li>• Managing such replicas in the face of updates introduces the problems of ensuring consistency among the replicas and cached copies.</li></ul>   |
| Execution models and frameworks ✓                            | <ul style="list-style-type: none"><li>• interleaving model and partial order model are two widely adopted models of distributed system executions ✓</li><li>• The input/output automata model [25] and the TLA (temporal logic of actions) are two other examples of models that provide different degrees of infrastructure ✓</li></ul>   |
| Program design and verification tools                        | <ul style="list-style-type: none"><li>• Methodically designed and verifiably correct programs can <u>greatly reduce the overhead of software design, debugging, and engineering</u></li><li>• Designing mechanisms to achieve these design and verification goals is a challenge.</li></ul>  |



## 2. Algorithmic Challenges ( CREPT-MGW)

| Challenge   | Description   |
|---|---|
| Time and global state<br><u>                    </u>  | <ol style="list-style-type: none"><li>1. The processes in the system are spread across three-dimensional physical space. Another dimension, <u>time</u>, has to be superimposed uniformly across space.</li><li>2. The challenges pertain to providing accurate physical time, and to providing a variant of time, called <u>logical time</u>.</li><li>3. Logical time is relative time, and eliminates the overheads of providing physical time for applications where physical time is not required.</li><li>4. Observing the global state of the system (across space) also involves the time dimension for consistent observation</li></ol> |
| Monitoring distributed events and predicates ✓        | <ul style="list-style-type: none"><li>• On-line algorithms for monitoring such predicates are hence important.</li><li>• <u>Event streaming</u> is used where streams of relevant events reported from different <u>processes</u> are examined collectively to detect predicates</li></ul>  |
| Graph algorithms and distributed routing algorithms ✓ | <ul style="list-style-type: none"><li>• The distributed system is modeled as a <u>distributed graph</u>, and the graph algorithms form the building blocks for a large number of <u>higher level communication</u>, data dissemination, object location, and object search functions.</li><li>• <u>the design of efficient distributed graph algorithms</u> is of paramount importance</li></ul>  |
| World Wide Web design ✓                               | <ul style="list-style-type: none"><li>• Minimizing response time to minimize user perceived latencies is important challenge</li><li>• Object search and navigation on the web efficiently is important</li></ul>   |

## 2. Algorithmic Challenges (Synch Mechanism)

- **Physical clock synchronization** Physical clocks ~~usually diverge~~ in their values due to hardware limitations. Keeping them synchronized is a fundamental challenge to maintain common time.
- **Leader election** All the processes need to agree on which process will play the role of a distinguished process – called a leader process. A leader is necessary even for many distributed algorithms because there is often some asymmetry – as in initiating some action like a broadcast or collecting the state of the system, or in “regenerating” a token that gets “lost” in the system.
- **Mutual exclusion** This is clearly a synchronization problem because access to the critical resource(s) has to be coordinated.
- **Deadlock detection and resolution** Deadlock detection should be coordinated to avoid duplicate work, and deadlock resolution should be coordinated to avoid unnecessary aborts of processes.
- **Termination detection** This requires cooperation among the processes to detect the specific global state of quiescence.
- **Garbage collection** Garbage refers to objects that are no longer in use and that are not pointed to by any other process. Detecting garbage requires coordination among the processes.

## 2. Algorithmic Challenges

1. **Wait-Free Algorithms**: Wait-freedom ensures a process can finish its task independently of others, vital for fault tolerance but often costly to design.
  2. **Mutual Exclusion**: Basic algorithms for mutual exclusion are taught in OS courses, but advanced methods like hardware primitives and wait-free algorithms will be covered here.
  3. **Register Constructions**: As future technologies like biocomputing and quantum computing challenge current hardware assumptions, register constructions explore concurrent access to memory without restrictions, forming a foundation for new architectures.
  4. **Consistency Models**: For multiple copies of data, various consistency levels trade off between coherence and implementation cost. Balancing strictness and efficiency is crucial.
-

## 2. Algorithmic Challenges (Fault Tolerant)

1. **Consensus Algorithms:** They help processes agree, even when some are malicious, by exchanging messages and handling misleading information.
2. **Replication and Replica Management:** Replication provides fault tolerance. We study advanced replication techniques for better efficiency.
3. **Voting and Quorum Systems:** These methods involve redundancy in the system and use voting based on quorum criteria for fault tolerance. The challenge is to design efficient algorithms.
4. **Distributed Databases and Commit:** In distributed databases, we aim to maintain traditional transaction properties. This field relates to ensuring message delivery guarantees in group communication during failures.
5. **Self-Stabilizing Systems:** These systems recover from errors and return to a good state. They require redundancy and efficient algorithms for self-stabilization.
6. **Checkpointing and Recovery Algorithms:** Periodic checkpointing records system states for recovery in case of failures. Coordination of checkpoints in distributed systems is a challenge.
7. **Failure Detectors:** In asynchronous systems, it's hard to tell if a process has failed. Failure detectors probabilistically suspect failure and converge on the status of suspected processes after timeouts.

## 2. Algorithmic Challenges (Load Balancing)

- **Data migration** The ability to move data (which may be replicated) around in the system, based on the access pattern of the users.
- **Computation migration** The ability to relocate processes in order to perform a redistribution of the workload.
- **Distributed scheduling** This achieves a better turnaround time for the users by using idle processing power in the system more efficiently.

G — U<sub>10000</sub>



## 2. Algorithmic Challenges (performance)

The following are some example issues arise in determining the performance:

- **Metrics** Appropriate metrics must be defined or identified for measuring the performance of theoretical distributed algorithms, as well as for implementations of such algorithms. The former would involve various complexity measures on the metrics, whereas the latter would involve various system and statistical metrics.
- **Measurement methods/tools** As a real distributed system is a complex entity and has to deal with all the difficulties that arise in measuring performance over a WAN/the Internet, appropriate methodologies and tools must be developed for measuring the performance metrics.



### 3. Applications of distributed computing

✓ **Mobile Systems:** In mobile systems, wireless communication is crucial, posing challenges like transmission range, power conservation, and interfacing with the wired internet. Computer science problems include routing, location management, channel allocation, and managing mobility.

There are two popular architectures:

1. **Base-Station Approach (Cellular):** Mobile processes in a cell communicate through a static base station, which presents graph-theoretical and engineering challenges.
2. **Ad-Hoc Network Approach:** No central base station; communication responsibilities are distributed among mobile nodes, requiring complex routing and presenting graph-theoretical and engineering challenges.

### 3. Applications of distributed computing

#### ✓ Sensor Networks:

1. Sensors, which can measure physical properties like temperature and humidity, have become affordable and are deployed in large numbers (over a million).
2. They report external events, not internal computer processes. These networks have various applications, including mobile or static sensors that communicate wirelessly or through wires. Self-configuring ad-hoc networks introduce challenges like position and time estimation.

#### Ubiquitous Computing:

1. Ubiquitous systems involve processors integrated into the environment, working in the background, like in sci-fi scenarios. Examples include smart homes and workplaces.
2. These systems are essentially distributed, use wireless tech, sensors, and actuators, and can self-organize. They often consist of many small processors in a dynamic network, connecting to more powerful resources for data processing.

# 3. Applications of distributed computing

## Peer-to-Peer (P2P) Computing:

1. In P2P computing, all processors interact as equals without any hierarchy, unlike client-server systems. P2P networks are often self-organizing and may lack a regular structure.
2. They don't use central directories for name resolution. Challenges include efficient object storage and lookup, dynamic reconfiguration, replication strategies, and addressing issues like privacy and security.

## Publish-Subscribe, Content Distribution, and Multimedia: (Netflix)

1. As information grows, we need efficient ways to distribute and filter it. Publish-Subscribe involves distributing information, letting users subscribe to what interests them, and then filtering it based on user preferences.
2. Content distribution is about sending data with specific characteristics to interested users, often used in web and P2P settings. When dealing with multimedia, we face challenges like large data, compression, and synchronization during storage and playback.

# 3. Applications of distributed computing

## Data Mining Algorithms:

1. They analyze large data sets to find patterns and useful information. For example, studying customer buying habits for targeted marketing.
2. This involves applying database and AI techniques to data. When data is distributed, as in private banking or large-scale weather prediction, efficient distributed data mining algorithms are needed.

## Security Challenges in Distributed Systems:

1. Traditional challenges include ensuring confidentiality, authentication, and availability.
2. The goal is efficient and scalable solutions.
3. In newer distributed architectures like wireless, peer-to-peer, grid, and pervasive computing, these challenges become more complex due to resource constraints, broadcast mediums, lack of structure, and network trust issues.



LIKE



COMMENT

SHARE



SUBSCRIBE