



# KTU NOTES

The learning companion.

**KTU STUDY MATERIALS | SYLLABUS | LIVE  
NOTIFICATIONS | SOLVED QUESTION PAPERS**

 Website: [www.ktunotes.in](http://www.ktunotes.in)



# **CST 402 - DISTRIBUTED COMPUTING**

# **Module - II**

# Module – II

## Lesson Plan

- **L1:** Logical time – A framework for a system of logical clocks, Scalar time
- **L2:** Vector time
- **L3:** Leader election algorithm – Bully Algorithm, Ring Algorithm.
- **L4:** Global state and snapshot recording algorithms – System model and definitions
- **L5:** Snapshot algorithm for FIFO channels – Chandy Lamport algorithm.
- **L6:** Termination detection – System model of a distributed computation
- **L7:** Termination detection using distributed snapshots.
- **L8 :** Termination detection by weight throwing, Spanning tree-based algorithm

# Logical time

- in distributed systems, it is not possible to have global physical time;
- it is possible to realize only an approximation of it
- As asynchronous distributed computations make progress in spurts, it turns out that the logical time, which advances in jumps, is sufficient to capture the fundamental monotonicity property<sub>(order)</sub> associated with causality in distributed systems
- Causality (or the causal precedence relation) among events in a distributed system is a powerful concept in reasoning, analysing, and drawing inferences about a computation
- The knowledge of the causal precedence relation among the events of processes helps solve a variety of problems in distributed systems

# Logical time

- Examples of some of these problems is as follows:

Distributed algorithms design

Tracking of dependent events

Knowledge about the progress

- The concept of causality is widely used by human beings, often unconsciously, in the planning, scheduling, and execution
- In day-to-day life, the global time to deduce causality relation is obtained from loosely synchronized clocks (i.e., wrist watches, wall clocks).
- However, in distributed computing systems, the rate of occurrence of events is several magnitudes higher and the event execution time is several magnitudes smaller.
- Consequently, if the physical clocks are not precisely synchronized, the causality relation between events may not be accurately captured

# Logical time

- Network Time Protocols which can maintain time accurate to a few tens of milliseconds on the Internet, are not adequate to capture the causality relation in distributed systems.
- However, in a distributed computation, generally the progress is made in spurts and the interaction between processes occurs in spurts
- In a system of logical clocks, every process has a logical clock that is advanced using a set of rules.
- Every event is assigned a timestamp and the causality relation between events can be generally inferred from their timestamps.
- The timestamps assigned to events obey the fundamental monotonicity property;
- that is, if an event  $a$  causally affects an event  $b$ , then the timestamp of  $a$  is smaller than the timestamp of  $b$ .

# A framework for a system of logical clocks

## Definition :

A system of logical clocks consists of a time domain  $T$  and a logical clock  $C$  .  
Elements of  $T$  form a partially ordered set over a relation  $<$

This relation is usually called the happened before or causal precedence.

Intuitively, this relation is analogous to the earlier than relation provided by the physical time.

The logical clock  $C$  is a function that maps an event  $e$  in a distributed system to an element in the time domain  $T$ , denoted as  $C(e)$  and called the timestamp of  $e$ , and is defined as follows:



# A framework for a system of logical clocks

$$C : H \mapsto T,$$

such that the following property is satisfied:

$$\text{for two events } e_i \text{ and } e_j, e_i \rightarrow e_j \implies C(e_i) < C(e_j).$$

This monotonicity property is called the *clock consistency condition*. When  $T$  and  $C$  satisfy the following condition,

$$\text{for two events } e_i \text{ and } e_j, e_i \rightarrow e_j \Leftrightarrow C(e_i) < C(e_j),$$

the system of clocks is said to be *strongly consistent*.

# Implementing logical clocks

Implementation of logical clocks requires addressing two issues

1. data structures local to every process to represent logical time
2. protocol (set of rules) to update the data structures to ensure the consistency condition.
  - A *local logical clock*, denoted by  $lc_i$ , that helps process  $p_i$  measure its own progress.
  - A *logical global clock*, denoted by  $gc_i$ , that is a representation of process  $p_i$ 's local view of the logical global time. It allows this process to assign consistent timestamps to its local events. Typically,  $lc_i$  is a part of  $gc_i$ .

The protocol ensures that a process's logical clock, and thus its view of the global time, is managed consistently. The protocol consists of the following two rules:

- **R1** This rule governs how the local logical clock is updated by a process when it executes an event (send, receive, or internal).
- **R2** This rule governs how a process updates its global logical clock to update its view of the global time and global progress. It dictates what information about the logical time is piggybacked in a message and how this information is used by the receiving process to update its view of the global time.

# Scalar Time

The scalar time representation was proposed by Lamport in 1978 as an attempt to totally order events in a distributed system

Time domain in this representation is the set of non-negative integers.

The logical local clock of a process  $p_i$  and its local view of the global time are squashed into one integer variable  $C_i$ .

Rules R1 and R2 to update the clocks are as follows:

# Scalar Time

Rules **R1** and **R2** to update the clocks are as follows:

- **R1** Before executing an event (send, receive, or internal), process  $p_i$  executes the following:

$$C_i := C_i + d \quad (d > 0).$$

In general, every time **R1** is executed,  $d$  can have a different value, and this value may be application-dependent. However, typically  $d$  is kept at 1 because this is able to identify the time of each event uniquely at a process, while keeping the rate of increase of  $d$  to its lowest level.

- **R2** Each message piggybacks the clock value of its sender at sending time. When a process  $p_i$  receives a message with timestamp  $C_{msg}$ , it executes the following actions:
  1.  $C_i := \max(C_i, C_{msg});$
  2. execute **R1**;
  3. deliver the message.

# Scalar Time

## Basic properties

### 1. Consistency property

Clearly, scalar clocks satisfy the monotonicity and hence the consistency property:

for two events  $e_i$  and  $e_j$ ,  $e_i \rightarrow e_j \implies C(e_i) < C(e_j)$ .

### 2. Total Ordering

- Scalar clocks can be used to totally order events in a distributed system
- The main problem in totally ordering events is that two or more events at different processes may have an identical timestamp.

# Scalar Time

- for two events

$$e_1 \text{ and } e_2, C(e_1) = C(e_2) \implies e_1 \parallel e_2.$$

- a tie-breaking mechanism is needed to order such events
- a tie is broken as follows:
  - process identifiers are linearly ordered and a tie among events with identical scalar timestamp is broken on the basis of their process identifiers.
- The lower the process identifier in the ranking, the higher the priority.
- The timestamp of an event is denoted by a tuple  $(t, i)$  where  $t$  is its time of occurrence and  $i$  is the identity of the process where it occurred.

# Scalar Time

## 3. Event counting

- By referring to “d” events can be counted , as its incremental with every instances

## 4. No strong consistency

The system of scalar clocks is not strongly consistent; that is, for two events  $e_i$  and  $e_j$

$$C(e_i) < C(e_j) \not\Rightarrow e_i \rightarrow e_j.$$

# Vector Time

The system of vector clocks was developed independently by Fidge , Mattern , and Schmuck .

In the system of vector clocks, the time domain is represented by a set of n-dimensional non-negative integer vectors.

A vector clock is a **data structure used for determining the partial ordering of events in a distributed system and detecting causality violations**

Vector Clocks are **used in a distributed systems to determine whether pairs of events are causally related**

Each process  $p_i$  maintains a vector  $vt_i[1..n]$ , where  $vt_i[i]$  is the local logical clock of  $p_i$  and describes the logical time progress at process  $p_i$ .



# Vector Time

- Initially all clocks are zero.
- Each time a process experiences an internal event, it increments its own **logical clock** in the vector by one. For instance, upon an event at process  $i$ , it updates  $VC_i[i] \leftarrow VC_i[i] + 1$ .
- Each time a process sends a message, it increments its own logical clock in the vector by one and then the message piggybacks a copy of vector.

# Vector Time

- **R1** Before executing an event, process  $p_i$  updates its local logical time as follows:

$$vt_i[i] := vt_i[i] + d \quad (d > 0).$$

- **R2** Each message  $m$  is piggybacked with the vector clock  $vt$  of the sender process at sending time.

Ktunotes.in

## Basic properties

### 1. Isomorphism

relation “ $\rightarrow$ ” induces a partial order on the set of events that are produced by a distributed execution.

If events in a distributed system are time stamped using a system of vector clocks, we have the following property.

# Vector Time

If two events  $x$  and  $y$  have timestamps  $vh$  and  $vk$ , respectively, then

$$x \rightarrow y \Leftrightarrow vh < vk$$

$$x \parallel y \Leftrightarrow vh \parallel vk.$$

## 2. Strong consistency

The system of vector clocks is strongly consistent; thus, by examining the vector timestamp of two events, we can determine if the events are causally related

## 3. Event counting

If  $d$  is always 1 in rule **R1**, then the  $i$ th component of vector clock at process  $p_i$ ,  $vt_i[i]$ , denotes the number of events that have occurred at  $p_i$  until that instant.

--

# Vector Time

## Applications

Since vector time tracks causal dependencies exactly, it finds a wide variety of applications.

- distributed debugging,
- implementations of causal ordering communication
- causal distributed shared memory,
- establishment of global breakpoints
- determining the consistency of checkpoints in optimistic recovery

## Leader election algorithm

- An algorithm for choosing a unique process to play a particular role (coordinator) is called an election algorithm.
- An election algorithm is needed for this choice.
- It is essential that all the processes agree on the choice.
- Afterwards, if the process that plays the role of server wishes to retire then another election is required to choose a replacement.
- We say that a process calls the election if it takes an action that initiates a particular run of the election algorithm.
- At any point in time, a process  $P_i$  is either a participant – meaning that it is engaged in some run of the election algorithm – or a non-participant – meaning that it is not currently engaged in any election.

Two algorithms,

- A ring-based election algorithm

- Bully algorithm

### 1. A ring-based election algorithm

Each process  $p_i$  has a communication channel to the next process in the ring,  $p_{(i+1) \bmod N}$ ,

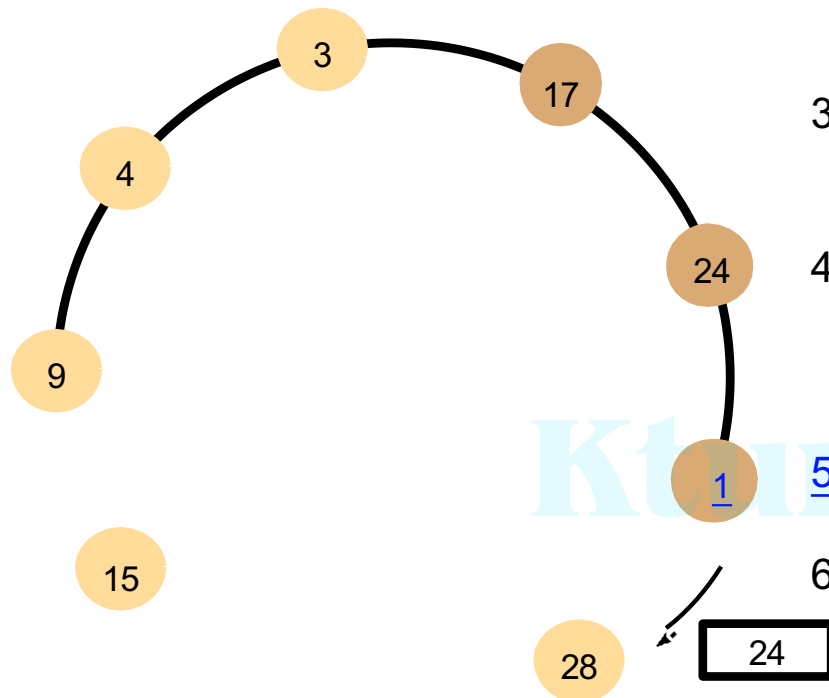
all messages are sent clockwise around the ring.

The goal of this algorithm is to elect a single process called the coordinator,

Initially, every process is marked as a non-participant in an election.

- Any process can begin an election. It proceeds by marking itself as a participant, placing its identifier in an election message and sending it to its clockwise neighbour.
- When a process receives an election message, it compares the identifier in the message with its own.
- If the arrived identifier is greater, then it forwards the message to its neighbour.
- If the arrived identifier is smaller and the receiver is not a participant, then it substitutes its own identifier in the message and forwards it; but it does not forward the message if it is already a participant.
- On forwarding an election message in any case, the process marks itself as a participant.
- If, however, the received identifier is that of the receiver itself, then this process's identifier must be the greatest, and it becomes the coordinator.
- The coordinator marks itself as a non-participant once more and sends an elected message to its neighbour, announcing its election and enclosing its identity

## A ring-based election in progress



1. **Initially**, every process is marked as non-participant. Any process can begin an election.
2. The **starting** process marks itself as participant and place its identifier in a message to its neighbour.
3. A process receives a message and **compare** it with its own. If the arrived identifier is **larger**, it passes on the message.
4. If arrived identifier is **smaller** and receiver is not a participant, substitute its own identifier in the message and forward if. It does not forward the message if it is already a participant.
5. On forwarding of any case, the process marks itself as a participant.
6. If the received identifier is that of the receiver itself, then this process's identifier must be the greatest, and it becomes the **coordinator**.
7. The coordinator marks itself as non-participant, set **elected<sub>i</sub>** and sends an **elected** message to its neighbour enclosing its ID.
8. When a process receives **elected** message, it marks itself as a non-participant, sets its variable **elected<sub>i</sub>** and forwards the message.



## 2. The bully algorithm

Process with highest id will be the coordinator

There are three types of message in this algorithm:

an election message is sent to announce an election;

an answer message is sent in response to an election message

a coordinator message is sent to announce the identity of the elected process.

The process that knows it has the highest identifier can elect itself as the coordinator simply by sending a coordinator message to all processes with lower identifiers.

On the other hand, a process with a lower identifier can begin an election by sending an election message to those processes that have a higher identifier and awaiting answer messages in response.

If none arrives within time  $T$ , the process considers itself the coordinator and sends a coordinator message to all processes with lower identifiers announcing this.

Otherwise, the process waits a further period  $T$  for a coordinator message to arrive from the new coordinator.

If a process  $p_i$  receives a coordinator message, it sets its variable  $elect_i$  to the identifier of the coordinator contained within it and treats that process as the coordinator.

If a process receives an election message, it sends back an answer message and begins another election – unless it has begun one already.

When a process, P, notices that the coordinator is no longer responding to requests, it initiates an election.

- P sends an ELECTION message to all processes with higher no.
- If no one responds, P wins the election and becomes a coordinator.

Ktunotes.in

- If one of the higher-ups answers, it takes over.

P' s job is done. When a process gets an ELECTION message from one of its lower-numbered colleagues:

- Receiver sends an OK message back to the sender to indicate that he is alive and will take over.

- Receiver holds an election, unless it is already holding one.
- Eventually, all processes give up but one, and that one is the new coordinator.
- The new coordinator announces its victory by sending all processes a message telling them that starting immediately it is the new coordinator.

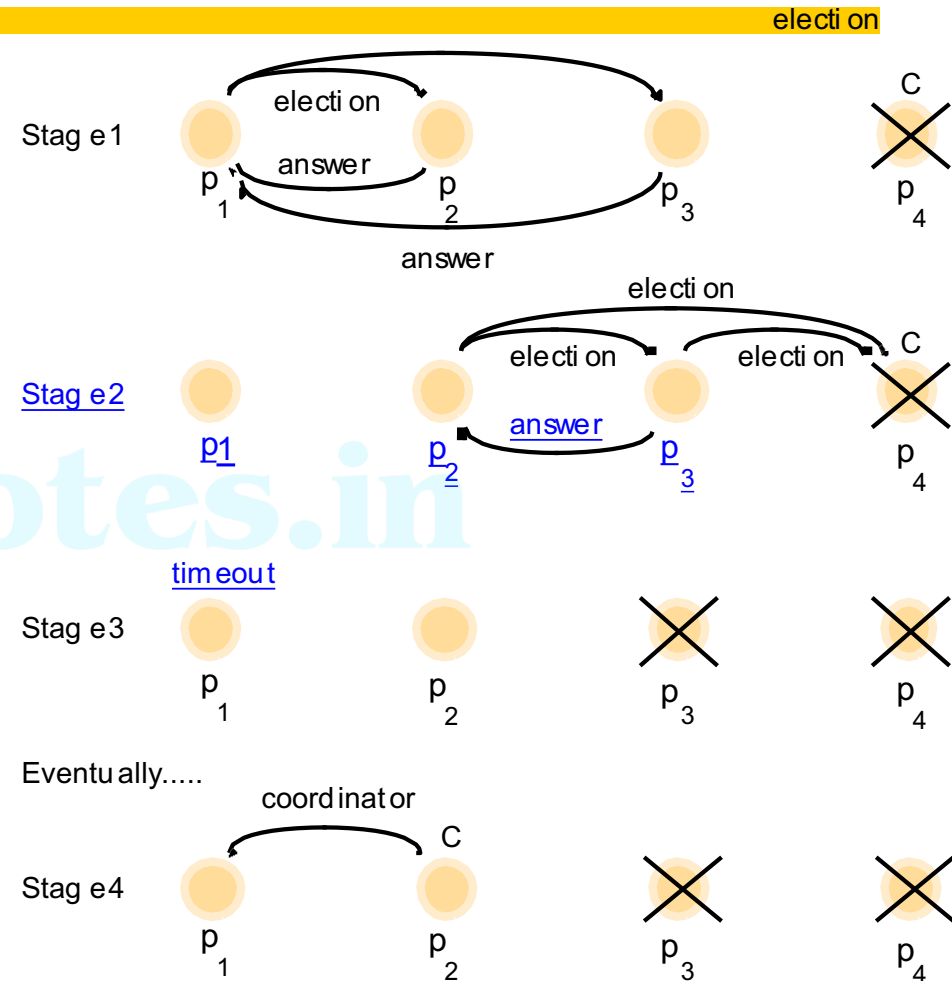
Ktunotes.in

If a process that was previously down comes back:

- It holds an election.
- If it happens to be the highest process currently running, it will win the election and take over the coordinator's job.
- Biggest guy" always wins and hence the name "bully" algorithm.

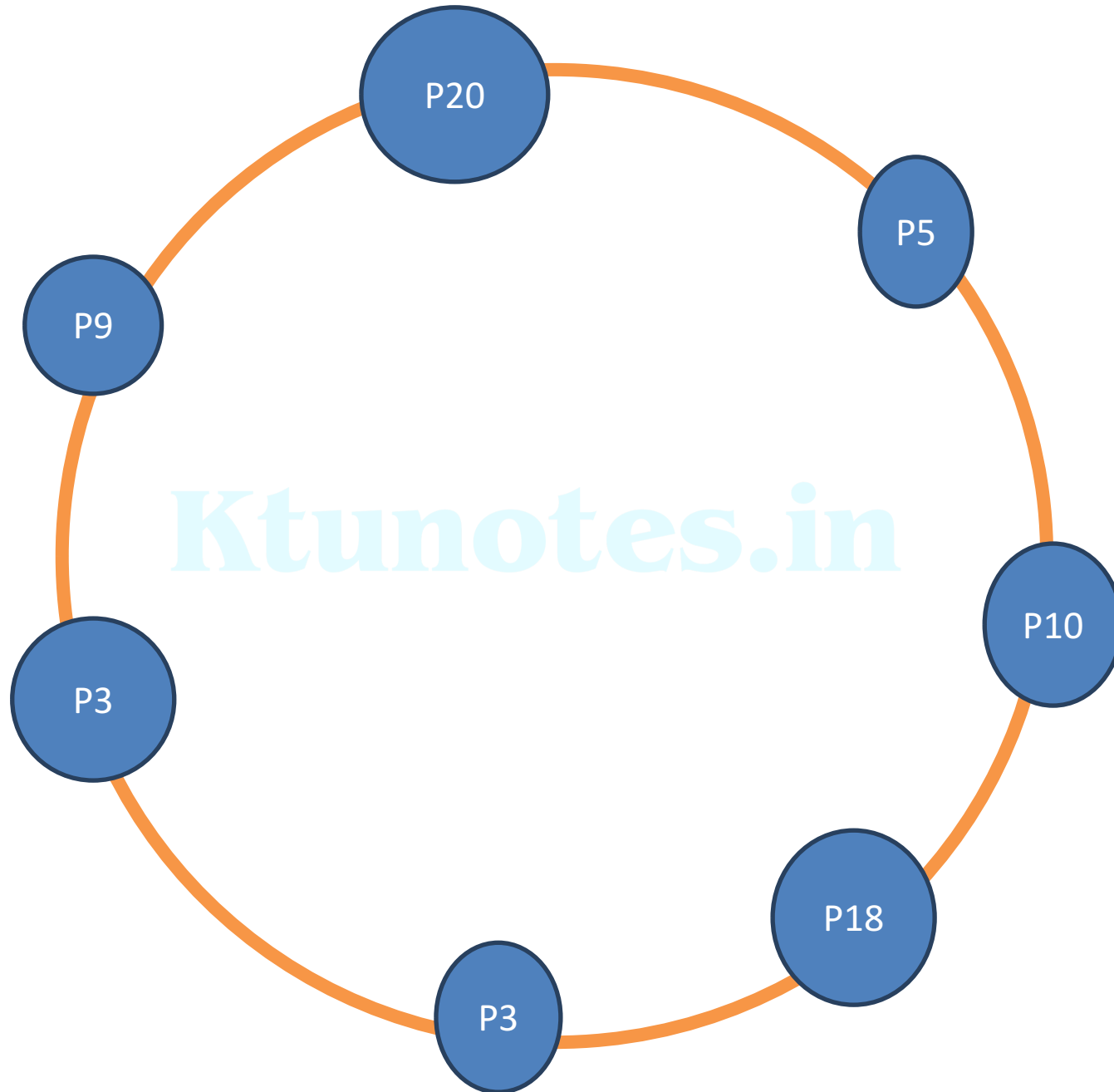
## The bully algorithm

1. The process begins an election by sending an election message to these processes that have a higher ID and awaits an answer in response.
2. If none arrives within time  $T$ , the process considers itself the coordinator and sends coordinator message to all processes with lower identifiers.
3. Otherwise, it waits a further time  $T'$  for coordinator message to arrive. If none, begins another election.
4. If a process receives a coordinator message, it sets its variable **elected<sub>i</sub>** to be the coordinator ID.
5. If a process receives an election message, it sends back an answer message and begins another election unless it has begun one already.



# Ring algorithm – work out

- In a ring topology 7 processes are connected with different ID's as shown: P20->P5->P10->P18->P3->P16->P9 If process P10 initiates election after how many message passes will the coordinator be elected and known to all the processes. What modification will take place to the election message as it passes through all the processes?
- Calculate total number of election messages and coordinator messages



# Bully Algorithm – Work out

- Pid's 0,4,2,1,5,6,3,7

P7 was the initial coordinator and crashed

Illustrate Bully algorithm, if P4 initiates election , Calculate total number of election messages and coordinator messages

Ktunotes.in