CST402 DISTRIBUTED COMPUTING-MODULE 2

PREPARED BY SHARIKA T R, ASSISTANT PROFESSOR, SNGCE

Course Outcome

CO1	Summarize various aspects of distributed computation model and logical time. **(Cognitive Knowledge Level: Understand)**
CO2	Illustrate election algorithm, global snapshot algorithm and termination detection algorithm. **(Cognitive Knowledge Level: Apply)**
CO3	Compare token based, non-token based and quorum based mutual exclusion algorithms. **(Cognitive Knowledge Level: Understand)**
CO4	Recognize the significance of deadlock detection and shared memory in distributed systems. **(Cognitive Knowledge Level: Understand)**
CO5	Explain the concepts of failure recovery and consensus. **(Cognitive Knowledge Level: Understand)**
CO6	Illustrate distributed file system architectures. **(Cognitive Knowledge Level: Understand)**

Mapping of course outcomes with program outcomes

	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
CO1	✓	✓	✓									✓
CO2	✓	✓	✓	✓								✓
CO3	✓	✓	✓									✓
CO4	✓	✓	✓									✓
CO5	✓	✓	✓									✓
CO6	✓	✓	✓									✓

Abstract POs defined by National Board of Accreditation			
PO#	**Broad PO**	**PO#**	**Broad PO**
PO1	Engineering Knowledge	PO7	Environment and Sustainability
PO2	Problem Analysis	PO8	Ethics
PO3	Design/Development of solutions	PO9	Individual and team work
PO4	Conduct investigations of complex problems	PO10	Communication
PO5	Modern tool usage	PO11	Project Management and Finance
PO6	The Engineer and Society	PO12	Life long learning

Assessment Pattern

Bloom's Category	Continuous Assessment Tests		End Semester Examination Marks (%)
	Test 1 (%)	Test 2 (%)	
Remember	30	30	30
Understand	50	50	50
Apply	20	20	20
Analyze			
Evaluate			
Create			

Mark Distribution

Total Marks	CIE Marks	ESE Marks	ESE Duration
150	50	100	3

Continuous Internal Evaluation Pattern:

Attendance	10 marks
Continuous Assessment Tests(Average of Internal Tests1&2)	25 marks
Continuous Assessment Assignment	15 marks

Module 2: Introduction

Syllabus- Election algorithm, Global state and Termination detection

Logical time – A framework for a system of logical clocks, Scalar time, Vector time.

Leader election algorithm – Bully algorithm, c. Global state and snapshot recording algorithms – System model and definitions, Snapshot algorithm for FIFO channels – Chandy Lamport algorithm.

Termination detection – System model of a distributed computation, Termination detection using distributed snapshots, Termination detection by weight throwing, Spanning-tree-based algorithm.

Introduction

The concept of causality between events is fundamental to the design and analysis of parallel and distributed computing and operating systems.

Usually causality is tracked using physical time.

In distributed systems, it is not possible to have a global physical time.

As asynchronous distributed computations make progress in spurts, the logical time is sufficient to capture the fundamental monotonicity property associated with causality in distributed systems.

three ways to implement logical time –

- scalar time,
- vector time, and
- matrix time. --- not needed

Causality among events in a distributed system is a powerful concept in reasoning, analyzing, 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, such as distributed algorithms design, tracking of dependent events, knowledge about the progress of a computation, and concurrency measures.

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

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 <.

Relation < is 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:

$$C : H \mapsto T,$$

such that the following property is satisfied:

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

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

- data structures local to every process to represent logical time and
- a protocol (set of rules) to update the data structures to ensure the consistency condition

Each process pi maintains data structures that allow it the following two capabilities:

- A local logical clock, denoted by lci, that helps process pi measure its own progress
- A logical global clock, denoted by gci, that is a representation of process pi's local view of the logical global time. It allows this process to assign consistent timestamps to its local events. Typically, lci is a part of gci.

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 aglobal time.

See you in next video

Scalar time

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 pi and its local view of the global time are squashed into one integer variable Ci.

Rules R1 and R2 to update the clocks

R1 Before executing an event (send, receive, or internal), process pi executes the following:

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

- 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 pi receives a message with timestamp Cmsg, it executes the following actions:

1. $C_i := max(C_i, C_{msg})$;
2. execute **R1**;
3. deliver the message.

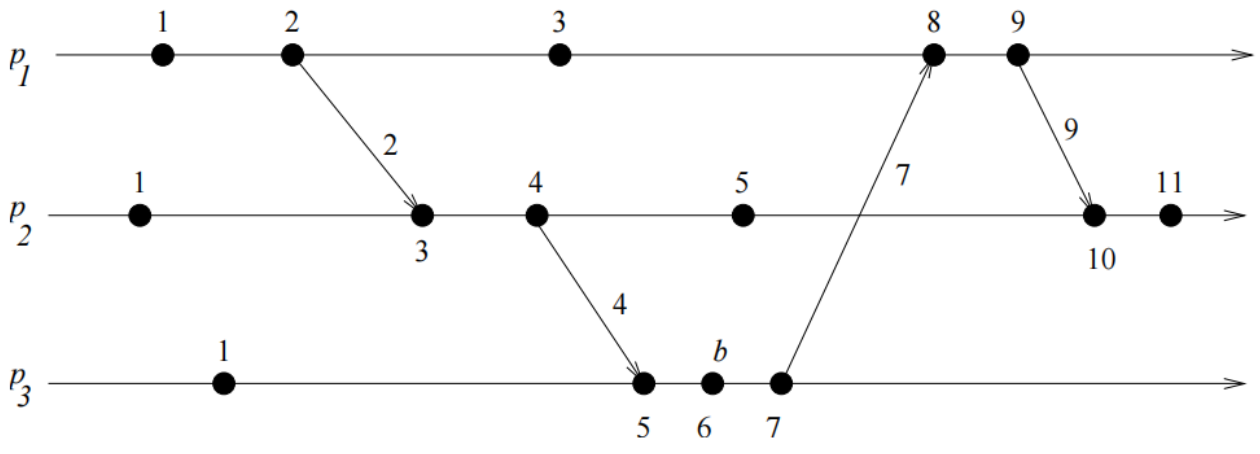


Figure 3.1: The space-time diagram of a distributed execution.

Scalar time basic Properties

1. Consistency property: Scalar clocks satisfy the monotonicity and hence the consistency property:

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

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 identical timestamp.
   - For example in Figure 3.1, the third event of process P1 and the second event of process P2 have identical scalar timestamp.

Total Ordering Cont..

A tie-breaking mechanism is needed to order such events.

A tie is broken as follows:

- Process identifiers are linearly ordered and 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.
- The total order relation $\prec$ on two events x and y with timestamps (h,i) and (k,j), respectively, is defined as follows:

$$x \prec y \Leftrightarrow (h < k \textbf{ or } (h = k \textbf{ and } i < j))$$

Since events that occur at the same logical scalar time are independent, they can be ordered using any arbitrary criterion without violating the causality relation →.

a total order is consistent with the causality relation "→".

$$x \prec y \Rightarrow (x \rightarrow y) \vee (x \,||\, y)$$

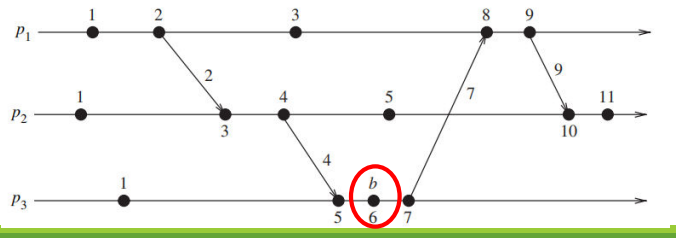
A total order is generally used to ensure liveness properties in distributed algorithms.

Requests are timestamped and served according to the total order based on these timestamps

Event counting

If the increment value d is always 1, the scalar time has the following interesting property:

- if event e has a timestamp h, then h−1 represents the minimum logical duration, counted in units of events, required before producing the event e ;
- we call it the **height of the event e**.
- In other words, h-1 events have been produced sequentially before the event e regardless of the processes that produced these events.
- For example, five events precede event b on the longest causal path ending at b.



No strong consistency

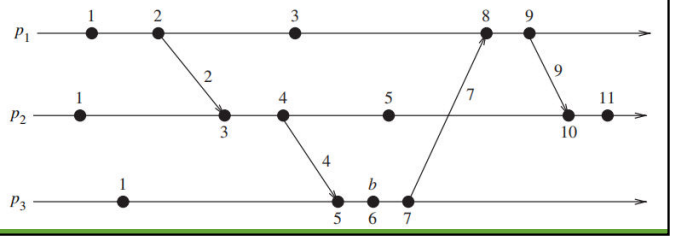
The system of scalar clocks is not strongly consistent; that is, for two events ei and ej

$$C(e_i) < C(e_j) \not\Longrightarrow e_i \rightarrow e_j$$

For eg,

- the third event of process P1 has smaller scalar timestamp than the third event of process P2.
- However, the former did not happen before the latter.

The reason that scalar clocks are not strongly consistent is that the logical local clock and logical global clock of a process are squashed into one, resulting in the loss causal dependency information among events at different processes



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.

Each process pi maintains a vector vti[1..n], where

$Vt_i[i]$→ is the local logical clock of pi and describes the logical time progress at process pi.

$Vt_i[j]$ → represents process pi's latest knowledge of process pj local time.

If $vt_i[j] = x$, then process pi knows that local time at process pj has progressed till x.

The entire vector vt_i constitutes p_i's view of the global logical time and is used to timestamp events.

Rules to update clock

Process pi uses the following two rules **R1** and **R2** to update its clock:

1. **R1** Before executing an event, process pi updates its local logical time as follows:

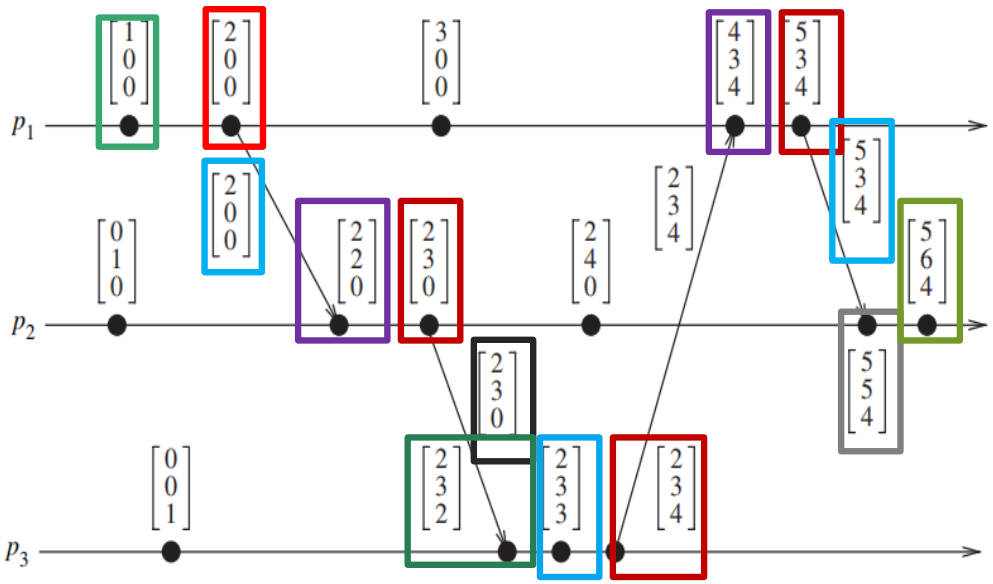
$$vt_i[i] := vt_i[i] + d \qquad (d > 0).$$

2. **R2** Each message *m* is piggybacked with the vector clock *vt* of the sender process at sending time. On the receipt of such a message *(m,vt)*, process pi executes the following sequence of actions:
   a. update its
   $$1 \le k \le n \ : \ vt_i[k] := max(vt_i[k], vt[k]);$$
   b. execute **R1**;
   c. deliver the message m.

an example of vector clocks progress with the increment value d = 1.

Initially, a vector clock is [0, 0, 0, 0, …, 0]

Figure 3.2 Evolution of vector time [19].



The following relations are defined to compare two vector timestamps, vh and vk:

$$vh = vk \Leftrightarrow \forall x : vh[x] = vk[x]$$

$$vh \leq vk \Leftrightarrow \forall x : vh[x] \leq vk[x]$$

$$vh < vk \Leftrightarrow vh \leq vk \text{ and } \exists x : vh[x] < vk[x]$$

$$vh \parallel vk \Leftrightarrow \neg(vh < vk) \wedge \neg(vk < vh).$$

Basic properties of Vector Time

Isomorphism

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

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

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

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

Thus, there is an isomorphism between the set of partially ordered events produced by a distributed computation and their vector timestamps.

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.

However, Charron-Bost showed that the dimension of vector clocks cannot be less than n, the total number of processes in the distributed computation, for this property to hold.

Event Counting

If d=1 (in rule R1), then the it^h component of vector clock at process pi,vti[i], denotes the number of events that have occurred at pi until that instant.

So, if an event e has timestamp vh, vh[j] denotes the number of events executed by process pj that causally precede e.

Clearly, $\sum vh[j] - 1$ represents the total number of events that causally precede e in the distributed computation.

Leader election algorithm

Leader election requires that all the processes agree on a common distinguished process, also termed as the *leader*.

A leader is required in many distributed systems because algorithms are typically not completely symmetrical, and some process has to take the lead in initiating the algorithm;

another reason is that we would not want all the processes to replicate the algorithm a initiation, to save on resources.

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 Pi 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 Leader election algorithms,

- A ring-based election algorithm
- Bully algorithm

Ring-based election algorithm

Each process p i has a communication channel to the next process in the ring, p (i + 1) mod 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.

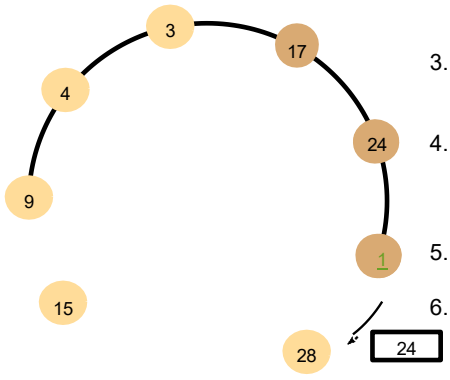
If the arrived identifier is smaller and the receiver is not a participant, then it substitutes its own identifier in the message and forwards it; but it does not forward the message if it is already a participant.

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 $\textbf{elected}_i$ 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 $\textbf{elected}_i$ and forwards the message.

The bully algorithm

Process with highest id will be the coordinator

There are three types of message in this algorithm:

1. an election message is sent to announce an election;
2. an answer message is sent in response to an election message
3. 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 pi receives a coordinator message, it sets its variable elected 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.
- If one of the higher-ups answers, it takes over.

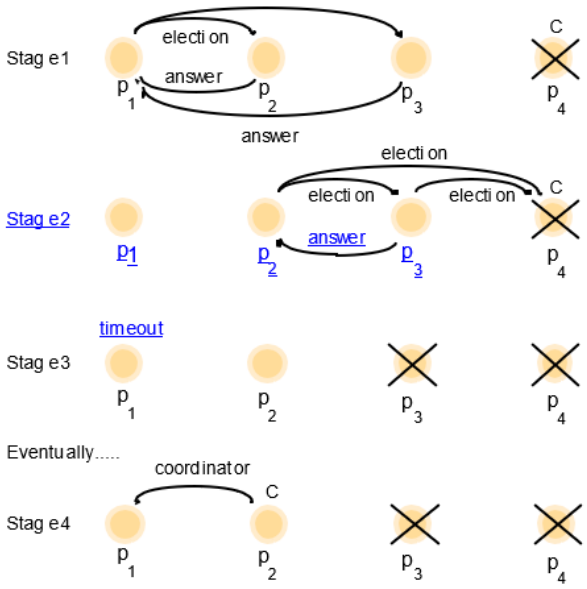
P' s job is done. When a process gets an ELECTION message from one of its lower-numbered colleagues:

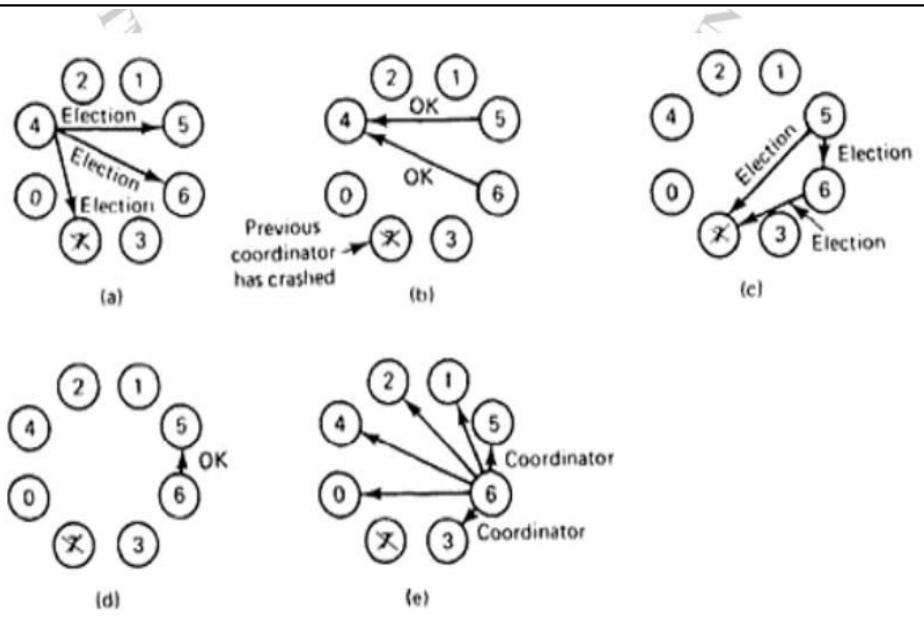
- Receiver sends an OK message back to the sender to indicate that he is alive and will take over.
- Receiver holds an election, unless it is already holding one.
- Eventually, all processes give up but one, and that one is the new coordinator.
- The new coordinator announces its victory by sending all processes a message telling them that starting immediately it is the new coordinator.

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.

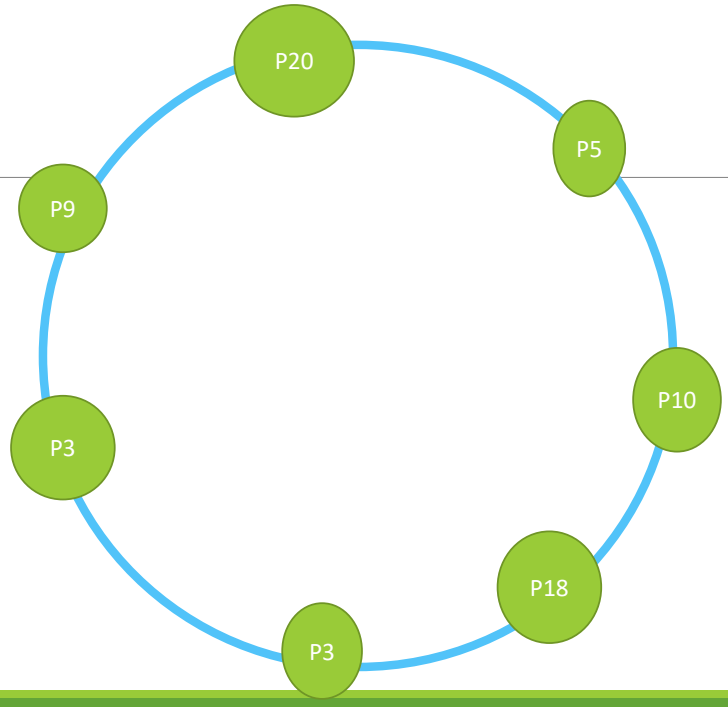
1. The process begins an election by sending an election message to these processes that have a higher ID and awaits an answer in response.
2. If none arrives within time T, the process considers itself the coordinator and sends coordinator message to all processes with lower identifiers.
3. Otherwise, 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 $elected_i$ 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



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