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

- Causality between events - fundamental to design and analysis of parallel n distributed computing and operating systems
- Causality -> usually tracked in physical time
- In distributed systems, not possible to have global physical time;
- Can realize only an approximation of it
- As **asynchronous distributed computations** make progress in spurts, the **logical time**, which advances in jumps, is sufficient to capture the ordering associated with **causality** in distributed systems

Logical time

- 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
- Causal precedence relation among the events of processes helps solve a variety of problems in distributed systems such as :

- Distributed algorithms design
  - Fair mutual exclusion algms, correct deadlock detection algms, consistency in replicated databases

- Knowledge about the progress : useful in discarding obsolete info, gc and termination detection

Logical time

- Tracking of dependent events
  - In debugging, helps to construct consistent states for resuming reexecution, helps build checkpoints in failure recovery, detection of inconsistencies in replicated databases

- Concurrency measure
  - How many events are causally related ->concurrency in the computation
  - All the events not causally related can be executed concurrently

Logical time

- 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).

- 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.

- In a distributed computation, generally the progress is made in spurts and the interaction between processes occurs in spurts

- Causality can be accurately captured by logical clocks

Logical clock

- 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**
- **3 ways to implement Logical time**
  - **1. Scalar time**
  - **2 .Vector time**
  - **3. Matrix time**

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.
  - 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:

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

A framework for a system of logical clocks

$$C : H \mapsto T,$$

such that the following property is satisfied:

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

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

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

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

Implementing logical clocks

- **Implementation of logical clocks -> two issues**
  - Data structures local to every process to represent logical time
  - A Protocol (set of rules) to update the data structures to ensure the consistency condition.

Implementing logical clocks contd…

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

- 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.

Implementing logical clocks contd…

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

- Scalar time representation -> proposed by Lamport in 1978
- 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 $\mathbf{p_i}$ and its local view of the global time are squashed into one integer variable $\mathbf{C_i}$.

- 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:

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

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

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

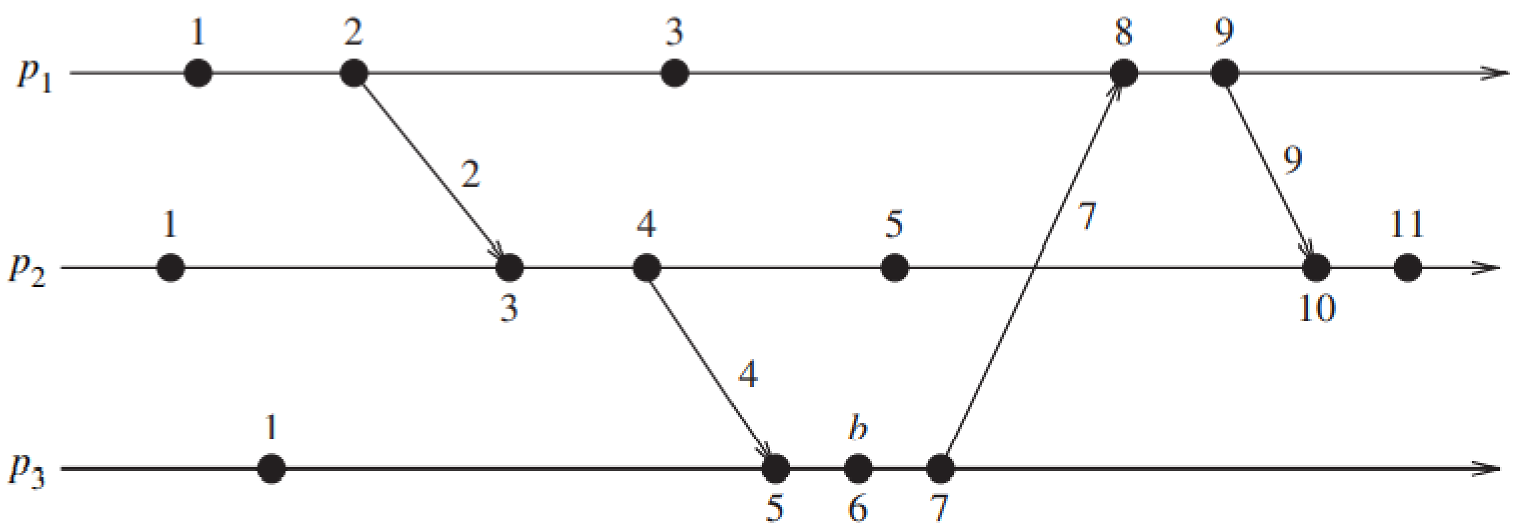
- **R1** Before executing an event (send, receive, or internal), process p_i executes the following:

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

- **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.

- Figure shows evolution of **scalar time** with d=1



Scalar Time

Basic properties

1. Consistency property

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

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

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 and e_2, $C(e_1) = C(e_2) \Longrightarrow e_1 \parallel e_2$.
- 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

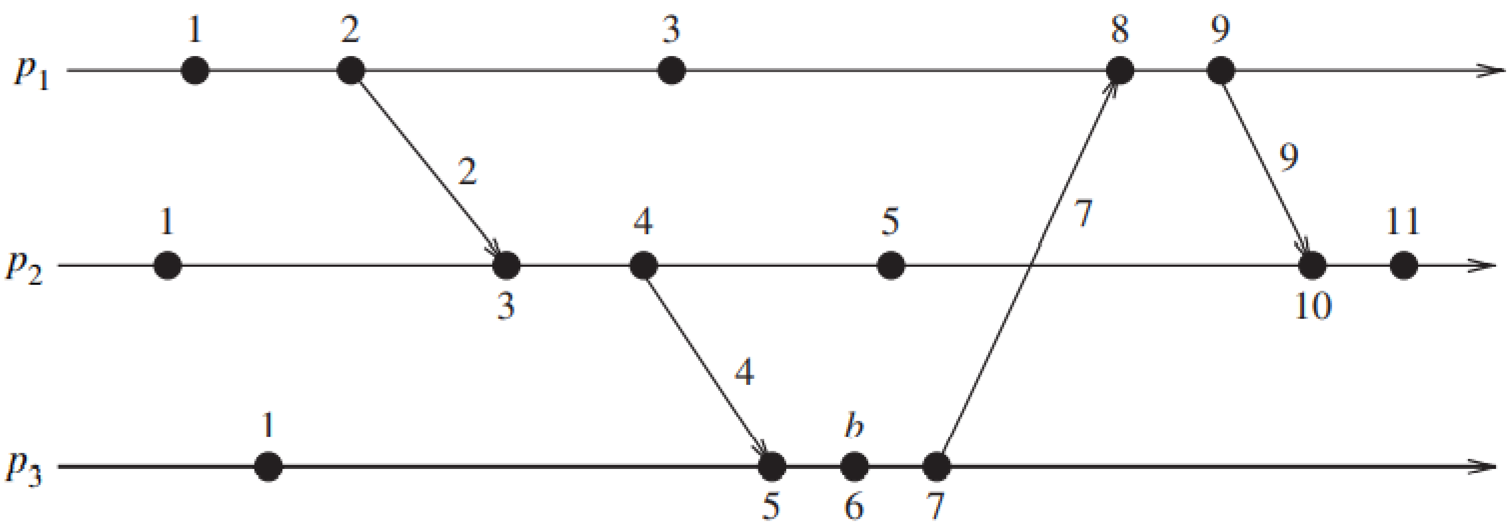
3. Event counting

- If the increment value $d = 1$, the scalar time has the following 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*.**
- i.e., *h-1* events have been produced sequentially before the event *e*

regardless of the processes that produced these events.

- For e.g. five events precede event *b* on the longest causal path ending at *b*.

Scalar Time



Scalar Time

4. No strong consistency

- The system of scalar clocks is not strongly consistent; that is, for two events e_i and e_j $$C(e_i) < C(e_j) \not\Longrightarrow e_i \rightarrow e_j.$$
- the third event of process P_1 has smaller scalar Timestamp than the third event of process P_2. 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.

- 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.

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. $vt_i[j]$ represents process p_i's latest knowledge of process p_j local time. If $vt_i[j] = x$, then process p_i knows that local time at process p_j 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.

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:

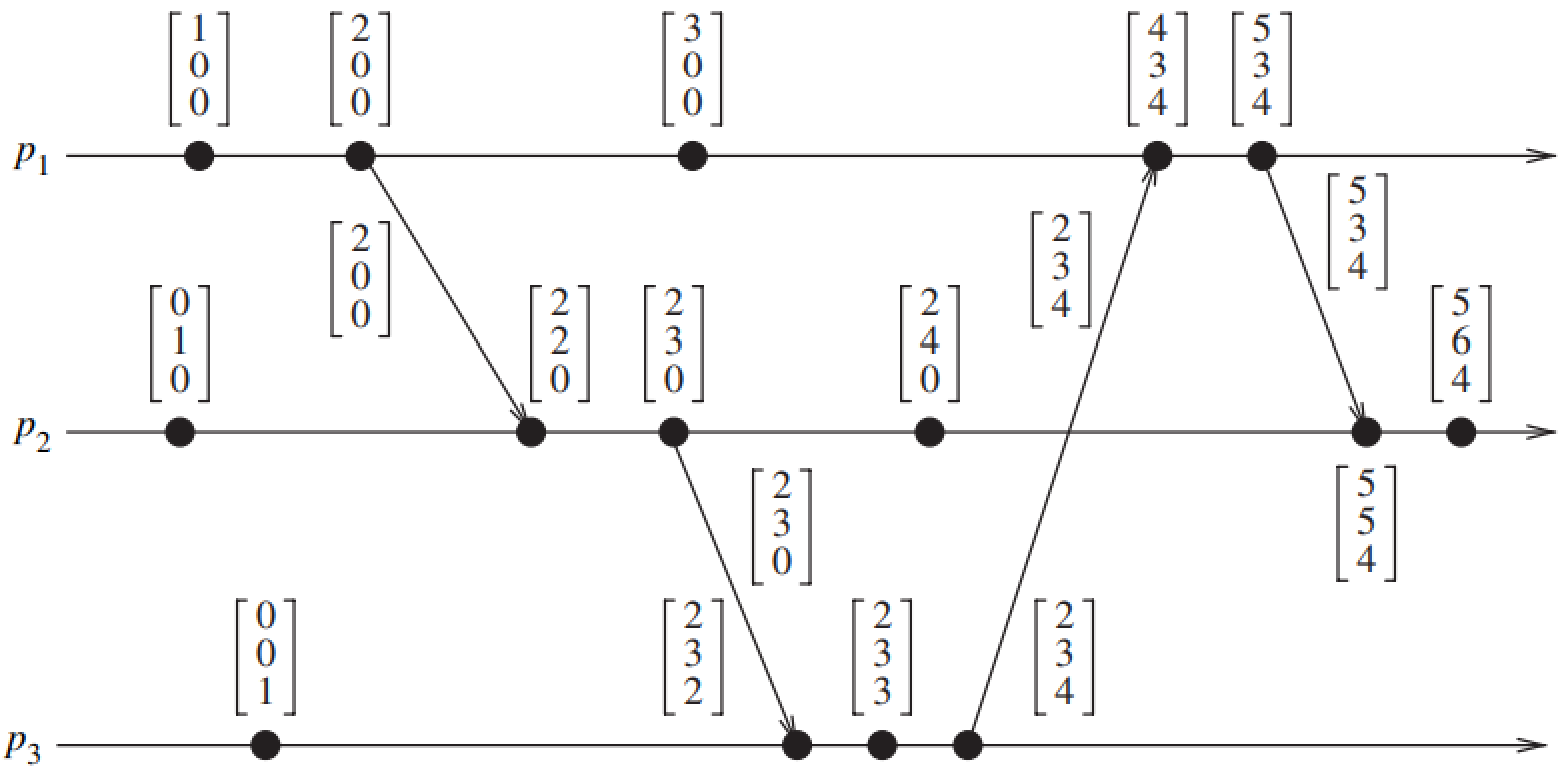
$$vt_i[i] := vt_i[i] + d \qquad (d > 0).$$

- **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 p_i executes the following sequence of actions:
  1. update its global logical time as follows:

$$1 \leq k \leq n \ : \ vt_i[k] := max(vt_i[k], vt[k]);$$

  2. execute **R1**;
  3. deliver the message m.

Vector Time



p_1
p_2
p_3
[1 0 0]
[2 0 0]
[3 0 0]
[4 3 4]
[5 3 4]
[2 0 0]
[2 3 4]
[5 3 4]
[0 1 0]
[2 2 0]
[2 3 0]
[2 4 0]
[5 6 4]
[2 3 0]
[5 5 4]
[0 0 1]
[2 3 2]
[2 3 3]
[2 3 4]

Basic properties
Vector Time

- **Isomorphism**
- relation "→" 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

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

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

If the process at which an event occurred is known, the test to compare two timestamps can be simplified as follows: if events x and y respectively occurred at processes p_i and p_j and are assigned timestamps vh and vk, respectively, then

$$x \rightarrow y \Leftrightarrow vh[i] \leq vk[i]$$

$$x \parallel y \Leftrightarrow vh[i] > vk[i] \wedge vh[j] < vk[j].$$

- **Strong consistency**
- The system of vector clocks is strongly consistent; i.e. 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 *ith* 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.

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

$$\sum vh[j] - 1$$

represents the total number of events that causally precede e in the distributed computation.

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.
- All the processes should agree on the choice.

- 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 Election Algorithms

- A ring-based election algorithm
- Bully algorithm

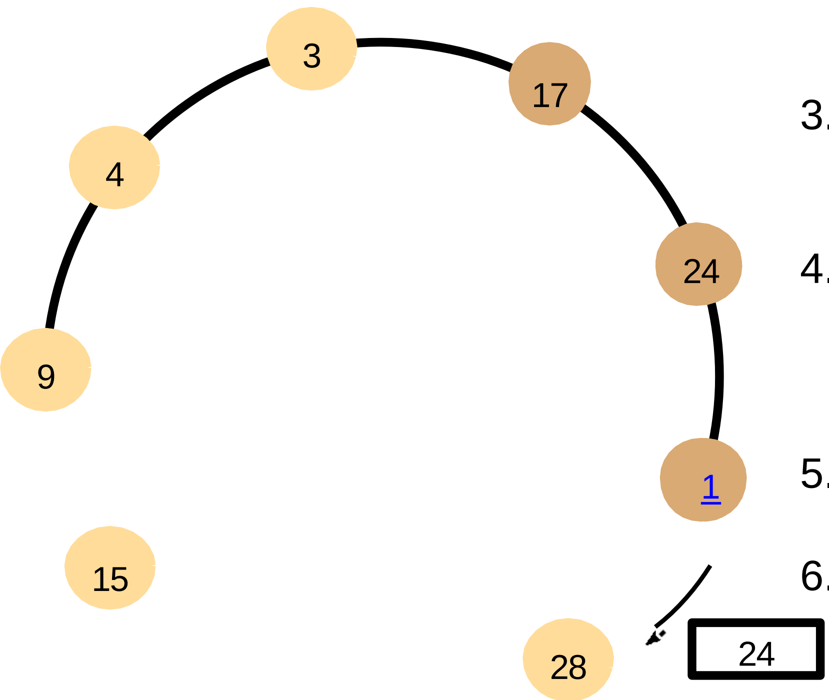
1. A ring-based election algorithm

- Each process p i has a communication channel to the next process in the ring, p (i + 1) 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, 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$_i$** 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 **elected$_i$** 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.
- 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 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:

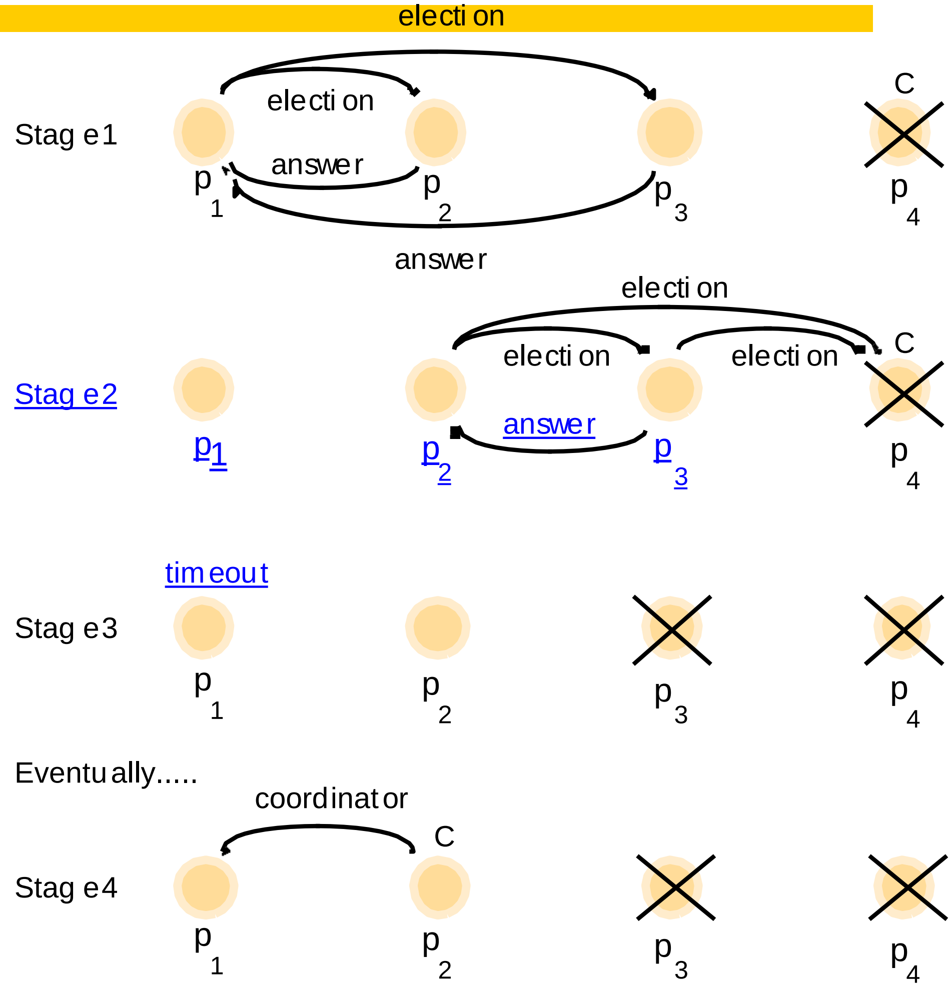
● It holds an election.

● 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.

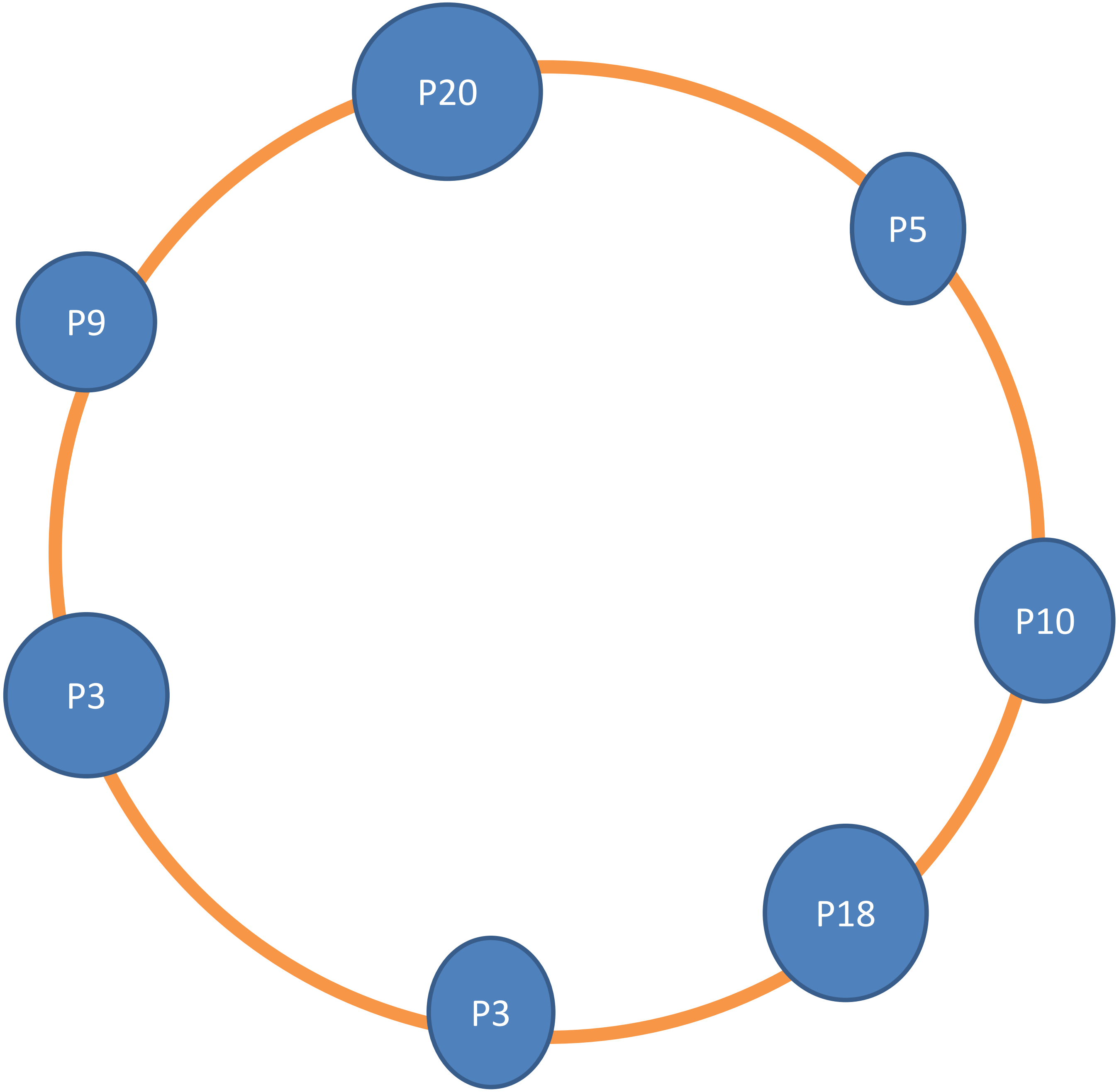
The bully algorithm

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**$_i$ 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


P20
P5
P9
P10
P3
P18
P3

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