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(order) 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:

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

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 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 $\mathbf{p_i}$ 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

Basic properties

1. Consistency property

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

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

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

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

- 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

- 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

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

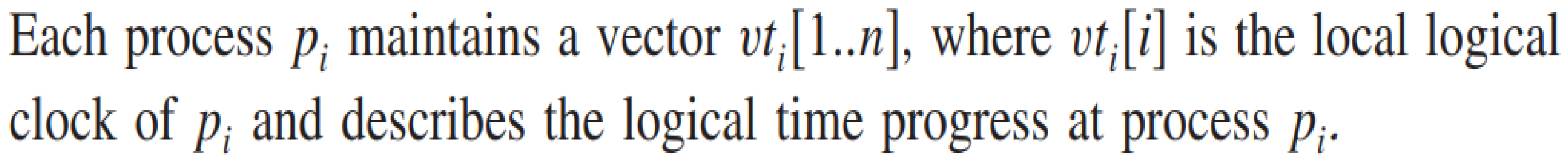
Vector Time

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:

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

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

Basic properties

1. 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

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

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 *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.

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 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 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.

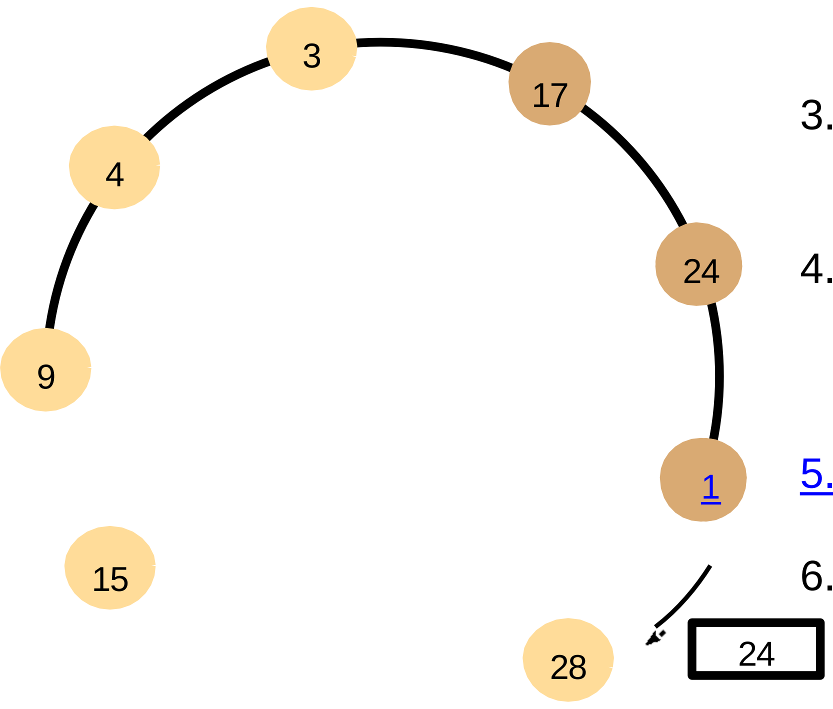
- If the arrived identifier is smaller and the receiver is not a participant, then it substitutes its own identifier in the message and forwards it; but it does not forward the message if it is already a participant.

- On forwarding an election message in any case, the process marks itself as a participant.

- If, 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$_i$** 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.

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 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:

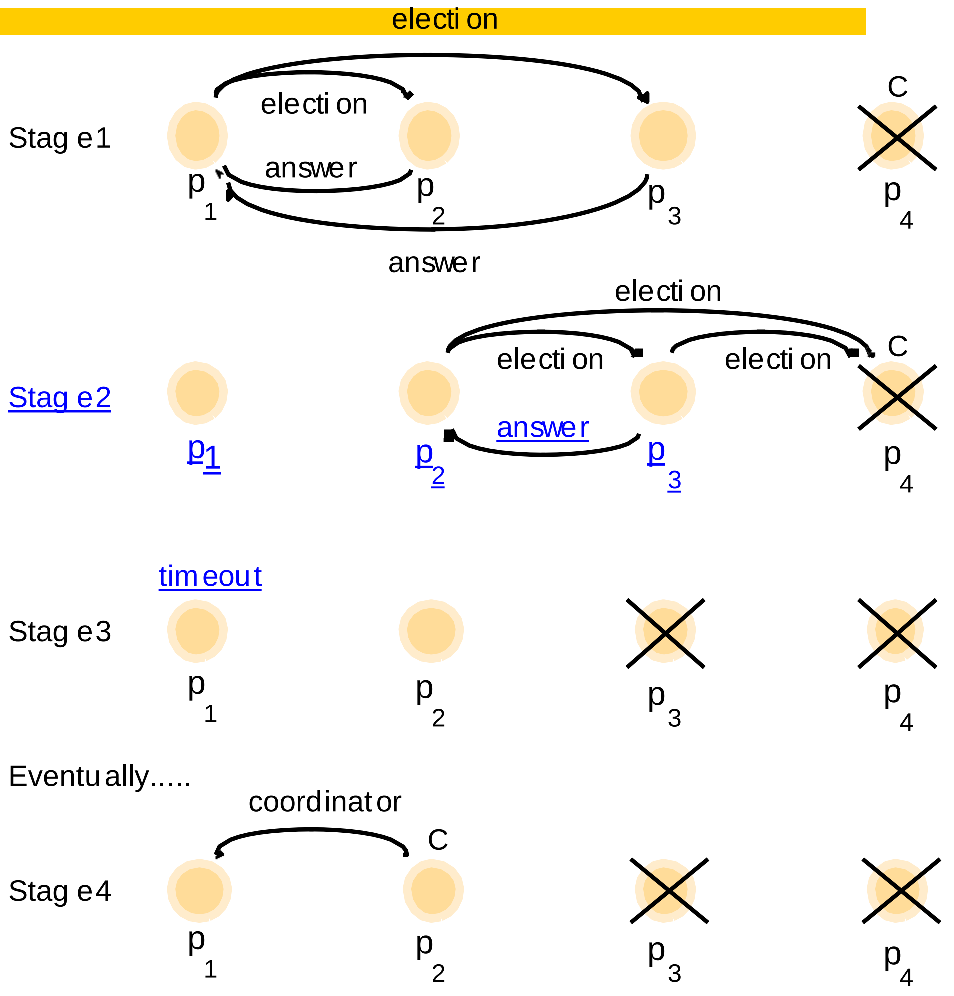
● It holds an election.

● If it happens to be the highest process currently running, it will win the election and take over the coordinator’ s job.

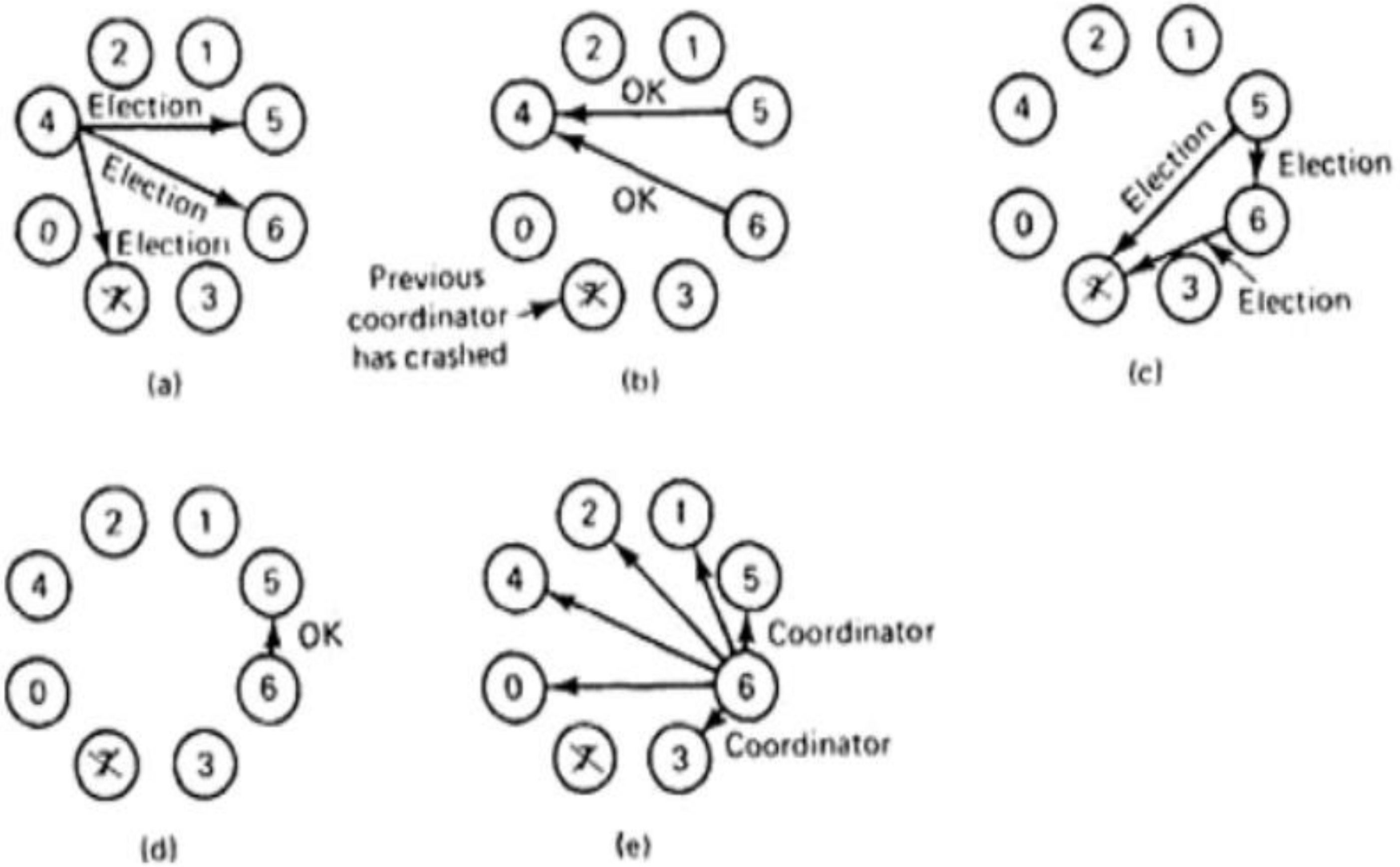
● Biggest guy” always wins and hence the name “ bully” algorithm.

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.

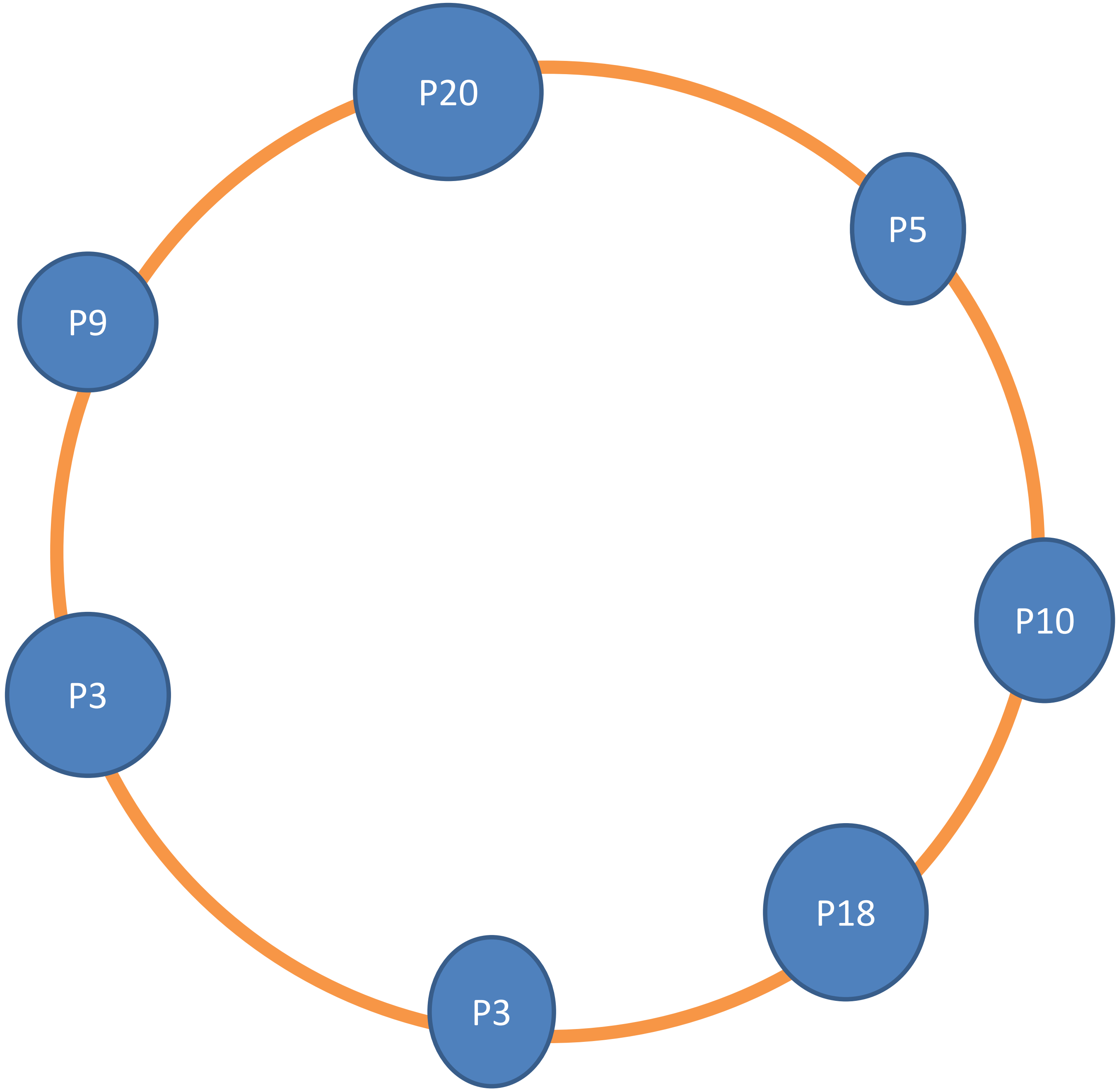


The bully algorithm



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
P10
P18
P3
P3
P9

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

Global state and snapshot recording algorithms

- Recording the global state of a distributed system is an important paradigm when one is interested in analyzing, testing, or verifying properties associated with distributed execution

- the lack of a globally shared memory and a global clock in a distributed system, added to the fact that message transfer delays in these systems are finite but unpredictable, makes this problem non-trivial.

Global state and snapshot recording algorithms

- A distributed computing system consists of spatially separated processes that do not share a common memory and communicate asynchronously with each other by message passing over communication channels

- Each component of a distributed system has a local state.

- The state of a process is characterized by the state of its local memory and a history of its activity.

- The state of a channel is characterized by the set of messages sent along the channel less the messages received along the channel

- The global state of a distributed system is a collection of the local states of its components

Applications-Recording Global state

- Detection of deadlocks, termination
- For Failure recovery: a global state of the distributed system (called a checkpoint) is periodically saved and recovery from a processor failure is done by restoring the system to the last saved global state

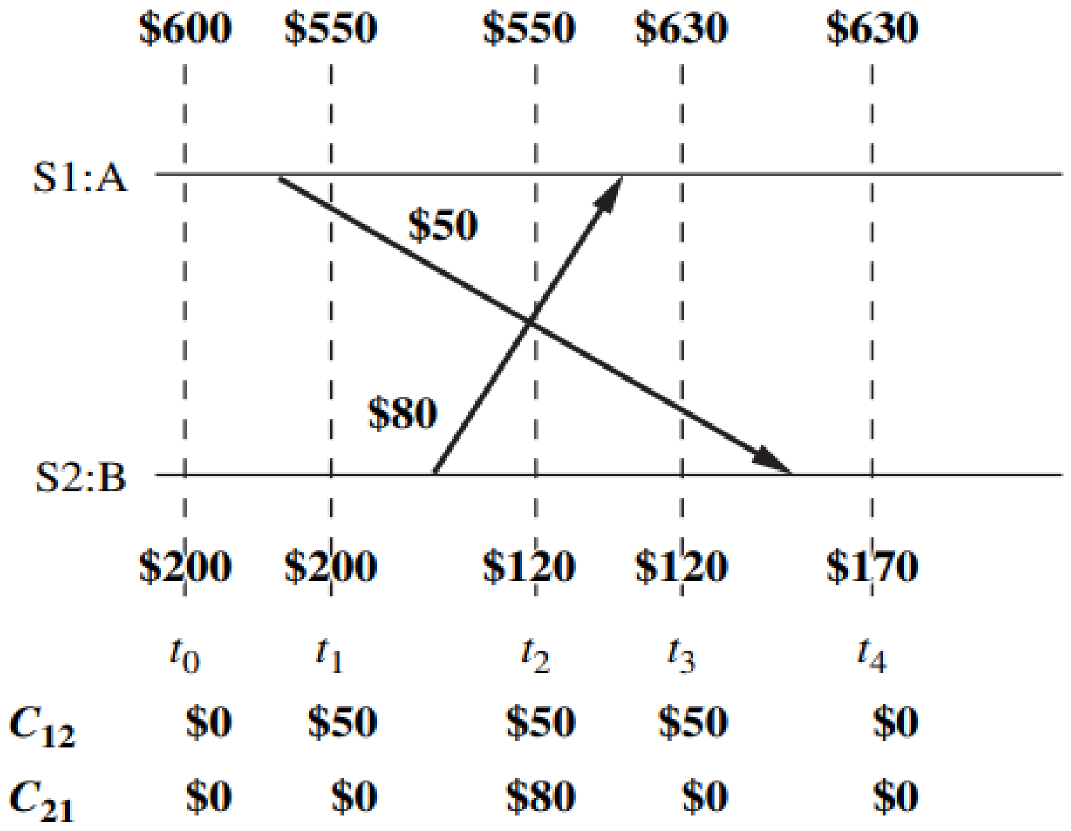
- for debugging distributed software
- monitoring distributed events such as in industrial process control,
- setting distributed breakpoints
- protocol specification and verification and
- discarding obsolete information

Global state and snapshot recording algorithms

Difficulties to record Global state

- If shared memory were available, an up-to-date state of the entire system would be available to the processes sharing the memory.
- The absence of shared memory necessitates ways of getting a coherent and **complete view of the system based on the local states of individual processes.**
- It is technologically infeasible to have perfectly synchronized clocks at various sites – clocks are bound to drift

A banking example to illustrate recording of consistent states.



- This simple example shows that recording a consistent global state of a distributed system is not a trivial task.
- Recording activities of individual components must be coordinated appropriately.

Global state and snapshot recording algorithms

- A meaningful global snapshot can be obtained if the components of the distributed system record their local states at the same time

System model

- The system consists of a collection of n processes, p1, p2, , pn, that are connected by channels.
- There is no globally shared memory and processes communicate solely by passing messages.
- There is no physical global clock in the system. Message send and receive is asynchronous.
- Messages are delivered reliably with finite but arbitrary time delay.
- The system can be described as a directed graph in which vertices represent the processes and edges represent unidirectional communication channels.

Global state and snapshot recording algorithms

System model

- Let C_{ij} denote the channel from process p_i to process p_j

- Processes and channels have states associated with them.

➢ The state of a process at any time is defined by the contents of processor registers, stacks, local memory

➢ The state of channel C_{ij}, denoted by Sc_{ij} ---→ **Set of messages in transit**

- The actions performed by a process are modeled as three types of events, namely, internal events, message send events, and message receive events.

- For a message m_{ij} that is sent by process p_i to process p_j, let **send(m_{ij})** and **rec(m_{ij})** denote its send and receive events, respectively.

- Occurrence of events changes the states of respective processes and channels, thus causing transitions in the global system state

Global state and snapshot recording algorithms

- For example, an internal event changes the state of the process at which it occurs.

- A send event (or a receive event) changes the state of the process that sends (or receives) the message and the state of the channel on which the message is sent (or received).

- The events at a process are linearly ordered by their order of occurrence.

- At any instant, the state of process **p_i**, denoted by **LS_i**, is a result of the sequence of all the events executed by pi up to that instant

- For an event e and a process state LSi , e∈LSi iff e belongs to the sequence of events that have taken process pi to state LSi .
- For an event e and a process state LSi , $e \notin LS_i$ iff e does not belong to the sequence of events that have taken process pi to state LSi .
- For a channel Cij , the following set of messages can be defined based on the local states of the processes pi and pj
- $\textbf{Transit}: transit(LS_i, LS_j) = \{m_{ij} \mid send(m_{ij}) \in LS_i \bigwedge rec(m_{ij}) \notin LS_j\}.$

Global state and snapshot recording algorithms

A consistent global state

- The global state of a distributed system is a collection of the local states of the processes and the channels. G lobal state GS is defined as

$$GS = \{\bigcup_i LS_i, \bigcup_{i,j} SC_{ij}\}.$$

- A global state GS is a **consistent global state** if it satisfies the following two conditions

$$\textbf{C1:}\ send(m_{ij}) \in LS_i \Rightarrow m_{ij} \in SC_{ij} \oplus rec(m_{ij}) \in LS_j \text{ (}\oplus\text{ is the Ex-OR operator).}$$

$$\textbf{C2:}\ send(m_{ij}) \notin LS_i \Rightarrow m_{ij} \notin SC_{ij} \wedge rec(m_{ij}) \notin LS_j.$$

Global state and snapshot recording algorithms

Interpretation in terms of cuts

- Cuts in a space–time diagram provide a powerful graphical aid in representing and reasoning about the global states of a computation.
- A cut is a line joining an arbitrary point on each process line that slices the space–time diagram into a PAST and a FUTURE.
- **A consistent global state corresponds to a cut in which every message received in the PAST of the cut has been sent in the PAST of that cut.**
- Such a cut is known as a consistent cut.
- All the messages that cross the cut from the PAST to the FUTURE are in **transit**, captured in the corresponding channel state.

Interpretation in terms of cuts

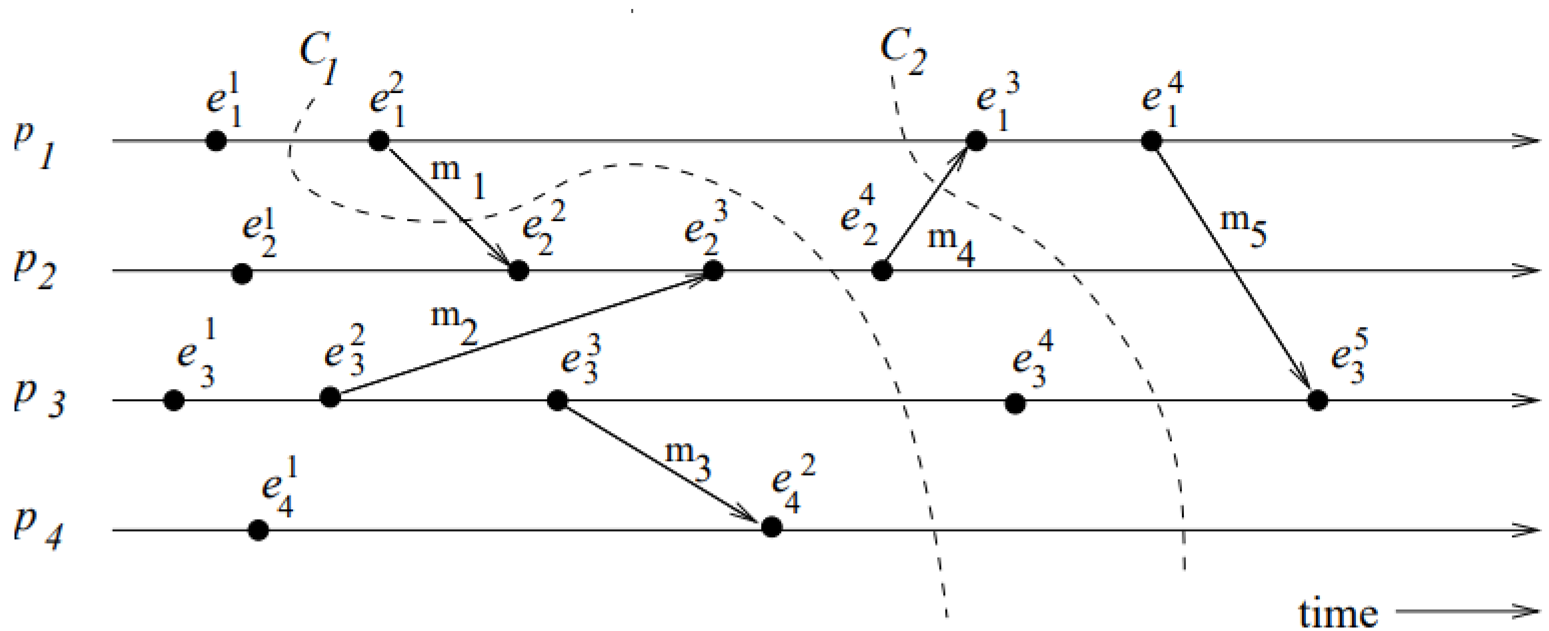


Figure 4.1: An Interpretation in Terms of a Cut.

- Cut C1 is inconsistent because message m1 is flowing from the FUTURE to the PAST.
- Cut C2 is consistent and message m4 must be captured in the state of channel C21.

Global state and snapshot recording algorithms

Issues in recording a global state

- If a global physical clock were available, the following simple procedure could be used to record a consistent global snapshot of a distributed system.

- In this, the initiator of the snapshot collection decides a future time at which the snapshot is to be taken and broadcasts this time to every process.

- All processes take their local snapshots at that instant in the global time.

- However, a global physical clock is not available in a distributed system and the following two issues need to be addressed in recording of a consistent global snapshot of a distributed system

- 1: How to distinguish between the messages to be recorded in the snapshot (either in a channel state or a process state) from those not to be recorded
- 2: How to determine the instant when a process takes its snapshot

Issues in recording a global state

The following two issues need to be addressed:

I1: How to distinguish between the messages to be recorded in the snapshot from those not to be recorded.

-Any message that is sent by a process before recording its snapshot, must be recorded in the global snapshot (from **C1**).
-Any message that is sent by a process after recording its snapshot, must not be recorded in the global snapshot (from **C2**).

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

-A process p_j must record its snapshot before processing a message m_{ij} that was sent by process p_i after recording its snapshot.

Snapshot algorithms for FIFO channels

- The Chandy-Lamport algorithm uses a control message, called a marker.
  - Computation messages –exchanged by underlying appln

- After a site has recorded its snapshot, it sends a marker along all of its outgoing channels before sending out any more messages.

- Since channels are FIFO, a marker separates the messages in the channel into those to be included in the snapshot (i.e., channel state or process state) from those not to be recorded in the snapshot.

The algorithm

A process initiates snapshot collection by executing the marker sending rule by which it records its local state and sends a marker on each outgoing channel

Chandy-Lamport algorithm

- The Chandy-Lamport algorithm uses a control message, called a marker whose role in a FIFO system is to separate messages in the channels.
- After a site has recorded its snapshot, it sends a marker, along all of its outgoing channels before sending out any more messages.
- A marker separates the messages in the channel into those to be included in the snapshot from those not to be recorded in the snapshot.
- A process must record its snapshot no later than when it receives a marker on any of its incoming channels.

- The algorithm can be initiated by any process by executing the “Marker Sending Rule” by which it records its local state and sends a marker on each outgoing channel.
- A process executes the “Marker Receiving Rule” on receiving a marker. If the process has not yet recorded its local state, it records the state of the channel on which the marker is received as empty and executes the “Marker Sending Rule” to record its local state.
- The algorithm terminates after each process has received a marker on all of its incoming channels.
- All the local snapshots get disseminated to all other processes and all the processes can determine the global state.

Snapshot algorithms for FIFO channels

Chandy–Lamport algorithm

- A process executes the marker receiving rule on receiving a marker.

- If the process has not yet recorded its local state, it records the state of the channel on which the marker is received as empty and executes the marker sending rule to record its local state Otherwise, the state of the incoming channel on which the marker is received is recorded

- The algorithm can be initiated by any process by executing the marker sending rule.
- The algorithm terminates after each process has received a marker on all of its incoming channels.

- The recorded local snapshots can be put together to create the global snapshot

Snapshot algorithms for FIFO channels

Chandy–Lamport algorithm

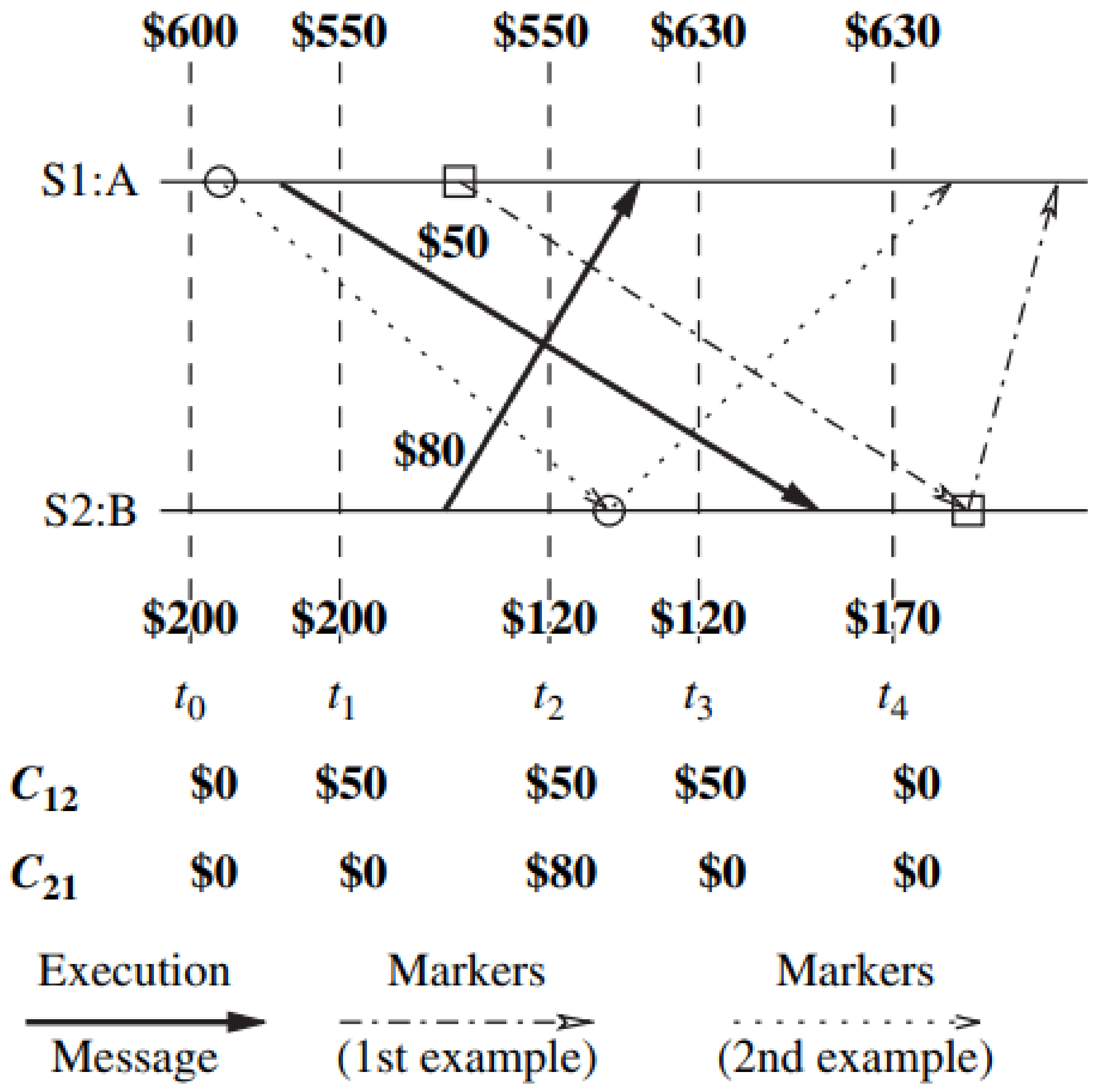
```
*Marker sending rule* for process $p_i$

(1) Process $p_i$ records its state.
(2) For each outgoing channel C on which a marker
    has not been sent, $p_i$ sends a marker along C
    before $p_i$ sends further messages along C.

*Marker receiving rule* for process $p_j$
On receiving a marker along channel C:
  **if** $p_j$ has not recorded its state **then**
     Record the state of C as the empty set
     Execute the “marker sending rule”
  **else**
     Record the state of C as the set of messages
     received along C after $p_{j's}$ state was recorded
     and before $p_j$ received the marker along C
```

Algorithm 4.1 The Chandy–Lamport algorithm.

Correctness :

- Due to FIFO property of channels, it follows that no message sent after the marker on that channel is recorded in the channel state. Thus, condition C2 is satisfied.
- When a process pj receives message mij that precedes the marker on channel Cij , it acts as follows: If process pj has not taken its snapshot yet, then it includes mij in its recorded snapshot. Otherwise, it records mij in the state of the channel Cij . Thus, condition C1 is satisfied.

Figure 4.3 Timing diagram of two possible executions of the banking example.



Snapshot algorithms for FIFO channels

- Site S1 records its local state (account A = \$550) and sends a marker to site S2.
- The marker is received by site S2 after t4. When site S2 receives the marker, it records its local state (account B = \$170), the state of channel C12 as \$0, and sends a marker along channel C21.
- When site S1 receives this marker, it records the state of channel C21 as \$80.
- The \$800 amount in the system is conserved in the recorded global state,
- A = \$550 B = \$170 C12 = \$0 C21 = \$80

Termination Detection

- In distributed processing systems, a problem is typically solved in a distributed manner with the cooperation of a number of processes.

- In such an environment, inferring if a distributed computation has ended is essential so that the results produced by the computation can be used

- A fundamental problem in distributed systems is to determine if a distributed computation has terminated.
- In the termination detection problem, a particular process (or all of the processes) must infer when the underlying computation has terminated

- messages used for the purpose of termination detection (by a termination detection algorithm) are called control messages.

.

Termination Detection

- there are two distributed computations taking place in the distributed system

➢ the underlying computation -> basic messages

➢ the termination detection algorithm ->control messages.

Termination Detection

A termination detection (TD) algorithm must ensure the following:

1. Execution of a TD algorithm cannot indefinitely delay the underlying computation; that is, execution of the termination detection algorithm must not freeze the underlying computation

2. The termination detection algorithm must not require addition of new communication channels between processes

Termination Detection

- A distributed computation is considered to be globally terminated if every process is locally terminated and there is no message in transit between any processes.

- A "locally terminated" state is a state in which a process has finished its computation and will not restart any action unless it receives a message.

- In the termination detection problem, a particular process (or all of the processes) must infer when the underlying computation has terminated

Termination Detection

System model of a distributed computation

- A distributed computation consists of a fixed set of processes that communicate solely by message passing.
- All messages are received correctly after an arbitrary but finite delay.
- Communication is asynchronous, i.e., a process never waits for the receiver to be ready before sending a message.
- Messages sent over the same communication channel may not obey the FIFO ordering.

Termination Detection

A distributed computation has the following characteristics:

1. At any given time during execution of the distributed computation, a process can be in only one of the two states**: active**, and **idle, (busy/passive)**

2. An active process can become idle at any time

3. An idle process can become active only on the receipt of a message from another process

4. Only active processes can send messages.

5. A message can be received by a process when the process is in either of the two states,

6. The sending of a message and the receipt of a message occur as atomic actions.

Termination Detection

System model of a distributed computation

Definition of termination detection

Let $p_i(t)$ denote the state (active or idle) of process p_i at instant t and $c_{i,j}(t)$ denote the number of messages in transit in the channel at instant t from process p_i to process p_j. A distributed computation is said to be terminated at time instant t_0 iff:

$$(\forall i :: p_i(t_0) = idle) \wedge (\forall i, j :: c_{i,j}(t_0) = 0).$$

Termination detection using distributed snapshots

- The algorithm uses the fact that a consistent snapshot of a distributed system captures stable properties.
- Termination of a distributed computation is a stable property.
- Thus, if a consistent snapshot of a distributed computation is taken after the distributed computation has terminated, the snapshot will capture the termination of the computation.
- The algorithm assumes that there is a logical bidirectional communication channel between every pair of processes.
- Communication channels are reliable but non-FIFO. Message delay is arbitrary but finite.

1. Termination detection using distributed snapshots

Informal description : The main idea behind the algorithm is as follows:

- when a computation terminates, there must exist a unique process which became idle last.

- When a process goes from active to idle, it issues a request to all other processes to take a local snapshot, and also requests itself to take a local snapshot.

- When a process receives the request, if it agrees that the requester became idle after itself, it grants the request by taking a local snapshot for the request.

- A request is said to be successful if all processes have taken a local snapshot for it.

- The requester or any external agent may collect all the local snapshots of a request.

Termination detection using distributed snapshots

Informal description

- If a request is successful, a global snapshot of the request can be obtained and the recorded state will indicate termination of the computation
- i.e in the recorded snapshot, all the processes are idle and there is no message in transit to any of the processes

Termination detection using distributed snapshots

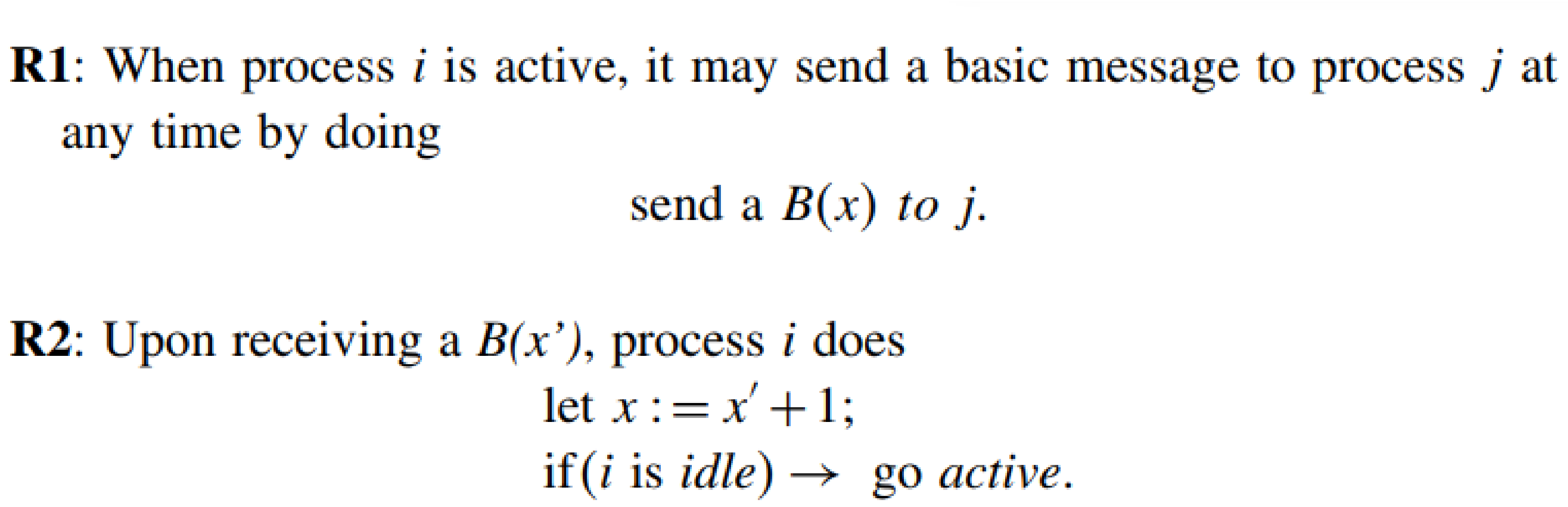
- The algorithm needs logical time to order the requests.
- Each process i maintains an logical clock denoted by x, which is initialized to zero at the start of the computation.
- A process increments its x by one each time it becomes idle.
- A basic message sent by a process at logical time x →B(x)
- A control message that requests processes to take local snapshot issued by process i at its logical time x → R(x, i).

Termination detection using distributed snapshots

- Each process synchronizes its logical clock x loosely with the logical clocks x's on other processes in such a way that it is the maximum of clock values ever received or sent in messages.
- Besides logical clock x, a process maintains a variable k such that when the process is idle, (x,k) is the maximum of the values (x, k) on all messages R(x, k) ever received or sent by the process.
- Logical time is compared as follows:
- $(x, k) > (x', k')$ iff $(x > x')$ or $((x = x')$ and $(k > k'))$
- i.e., a tie between x and x' is broken by the process identification numbers k and k'.

Termination detection using distributed snapshots

- The algorithm is defined by the following **four rules**



R1: When process i is active, it may send a basic message to process j at any time by doing

send a $B(x)$ *to* j.

R2: Upon receiving a $B(x')$, process i does

let $x := x' + 1$;

if(i is *idle*) $\rightarrow$ go *active*.

Termination detection using distributed snapshots

R3: When process i goes *idle*, it does

```
        let $x := x + 1$;
        let $k := i$;
        send message $R(x, k)$ to all other processes;
        take a local snapshot for the request by $R(x, k)$.
```

R4: Upon receiving message $R(x', k')$, process i does

```
    [$((x', k') > (x, k)) \wedge (i$ is idle$) \rightarrow$ let$(x, k) := (x', k')$;
                      take a local snapshot for the request by$R(x', k')$;
    □
    $((x', k') \leq (x, k)) \wedge (i$ is idle$) \rightarrow$ do nothing;
    □
    $(i$ is active$) \rightarrow$ let $x := max(x', x)$].
```

Termination detection using distributed snapshots

- R1 -> when a process sends a basic message to any other process, it sends its logical clock value in the message
- R2 ->When a process receives a basic message, it updates its logical clock based on the clock value contained in the message

Termination detection using distributed snapshots

- Rule R3 → when a process becomes idle, it updates its local clock, sends a request for snapshot R(x, k) to every other process, and takes a local snapshot for this request.
- R4 →On the receipt of a message R(x , k), the process takes a local snapshot if it is idle and (x , k) > (x, k), i.e., timing in the message is later than the local time at the process →implying that the sender of R(x , k) terminated after this process

Termination detection using distributed snapshots

- ➢ (x , k) ≤ (x, k), implying that the sender of R(x , k) terminated before this process.
- Hence, the sender of R(x , k) cannot be the last process to terminate. Thus, the receiving process does not take a snapshot for it.
- ➢ In the third case, the receiving process has not even terminated. Hence, the sender of R(x , k) cannot be the last process to terminate and no snapshot is taken.
- ➢ The last process to terminate will have the largest clock value. Every process will take a snapshot for it; it will not take a snapshot for any other process

2. Termination detection by weight throwing

- In termination detection by weight throwing, a process called controlling agent monitors the computation.
- A communication channel exists between each of the processes and the controlling agent and also between every pair of processes.
- Initially, all processes are in the idle state.
- The weight at each process is zero and the weight at the controlling agent is 1.
- The computation starts when the controlling agent sends a basic message to one of the processes.
- The process becomes active and the computation starts.
- A non-zero weight W ($0 < W \leq 1$) is assigned to each process in the active state and to each message in transit in the following manner:

2. Termination detection by weight throwing

- When a process sends a message, it sends a part of its weight in the message.
- When a process receives a message, it add the weight received in the message to its weight.
- Thus, the sum of weights on all the processes and on all the messages in transit is always 1.
- When a process becomes passive, it sends its weight to the controlling agent in a control message, which the controlling agent adds to its weight.
- The controlling agent concludes termination if its weight becomes 1.

2. Termination detection by weight throwing

Notation

- The weight on the controlling agent and a process is in general represented by W.
- $B(DW)$: A basic message B is sent as a part of the computation, where DW is the weight assigned to it.
- $C(DW)$: A control message C is sent from a process to the controlling agent where DW is the weight assigned to it.

2. Termination detection by weight throwing

Formal description

The algorithm is defined by the following four rules

Rule 1: The controlling agent or an active process may send a basic message to one of the processes, say P, by splitting its weight W into $W1$ and $W2$ such that $W1+W2=W$, $W1>0$ and $W2>0$. It then assigns its weight $W:=W1$ and sends a basic message $B(DW:=W2)$ to P.

Rule 2: On the receipt of the message $B(DW)$, process P adds DW to its weight W $(W:=W+DW)$. If the receiving process is in the idle state, it becomes active.

Rule 3: A process switches from the active state to the idle state at any time by sending a control message $C(DW:=W)$ to the controlling agent and making its weight $W:=0$.

Rule 4: On the receipt of a message $C(DW)$, the controlling agent adds DW to its weight $(W:=W+DW)$. If W $=1$, then it concludes that the computation has terminated.

3. A spanning-tree-based termination detection algorithm

- The algorithm assumes there are N processes Pi, 0 ≤ i ≤ N-1, which are modelled as the nodes i, 0 ≤ i ≤ N-1, of a fixed connected undirected graph.
- The edges of the graph represent the communication channels, through which a process sends messages to neighbouring processes in the graph.
- The algorithm uses a fixed spanning tree of the graph with process P0 at its root which is responsible for termination detection
- Process P0 communicates with other processes to determine their states and the messages used for this purpose are called signals.
- All leaf nodes report to their parents, if they have terminated.
- A parent node will similarly report to its parent when it has completed processing and all of its immediate children have terminated, and so on.
- The root concludes that termination has occurred, if it has terminated and all of its immediate children have also terminated

3. A spanning-tree-based termination detection algorithm

- The termination detection algorithm generates two waves of signals moving inward and outward through the spanning tree.

- Initially, a contracting wave of signals, called tokens, moves inward from leaves to the root.

- If this token wave reaches the root without discovering that termination has occurred, the root initiates a second outward wave of repeat signals.

- As this repeat wave reaches leaves, the token wave gradually forms and starts moving inward again.

- This sequence of events is repeated until the termination is detected

3. A spanning-tree-based termination detection algorithm

Definitions

1. Tokens: a contracting wave of signals that move inward from the leaves to the root.
2. Repeat signal: if a token wave fails to detect termination, node *P*0 initiates another round of termination detection by sending a signal called Repeat, to the leaves.
3. The nodes which have one or more tokens at any instant form a set *S*.
4. A node *j* is said to be outside of set *S* if *j* does not belong to *S* and the path (in the tree) from the root to *j* contains an element of *S*. Every path from the root to a leaf may not contain a node of *S*.
5. Note that all nodes outside *S* are idle. This is because, any node that terminates, transmits a token to its parent. When a node transmits the token, it goes out of the set *S*.

3. A spanning-tree-based termination detection algorithm

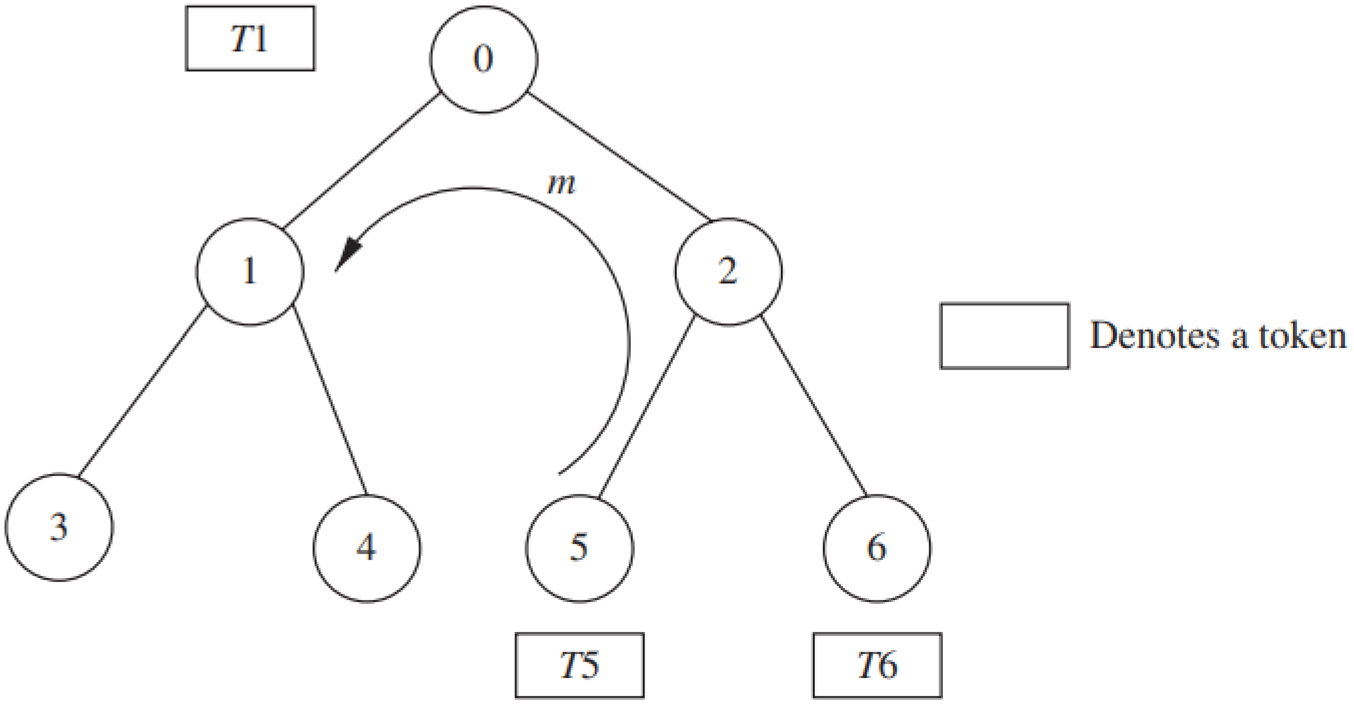
- Initially, each leaf process is given a token.
- Each leaf process, after it has terminated, sends its token to its parent.
- When a parent process terminates and after it has received a token from each of its children, it sends a token to its parent.
- This way, each process indicates to its parent process that the subtree below it has become idle.
- In a similar manner, the tokens get propagated to the root.
- The root of the tree concludes that termination has occurred, after it has become idle and has received a token from each of its children.

A problem with the algorithm

- This simple algorithm fails under some circumstances. After a process has sent its token to its parent, it should remain idle.
- The problem arises when a process after it has sent a token to its parent, receives a message from some other process and becomes active again.
- The root node just because it received a token from a child, can’t conclude that all processes in the child’s subtree have terminated.

A spanning-tree-based termination detection algorithm

- An Example of the problem



A spanning-tree-based termination detection algorithm :basic Idea

- Main idea is to color the processes and tokens and change the color when messages such as in Figure 1 are involved.
- The root can determine that an idle process has been activated by a message, based on the color of the token it receives from its children.
- All tokens are initialized to white.
- If a process had sent a message to some other process, it sends a black token to its parent on termination; otherwise, it sends a white token on termination.
- This gets propagated and finally the root node knows that message-passing was involved when it receives a black token from one of its children
- the root asks all nodes in the system to restart the termination detection. Repeat signal
- all leaves will restart the termination detection algorithm

3. A spanning-tree-based termination detection algorithm

The algorithm description

The algorithm works as follows:

1. Initially, each leaf process is provided with a token. The set S is used for book-keeping to know which processes have the token. Hence S will be the set of all leaves in the tree.
2. Initially, all processes and tokens are white. As explained above, coloring helps the root know if a message-passing was involved in one of the subtrees.
3. When a leaf node terminates, it sends the token it holds to its parent process.
4. A parent process will collect the token sent by each of its children. After it has received a token from all of its children and after it has terminated, the parent process sends a token to its parent.
5. A process turns black when it sends a message to some other process. This coloring scheme helps a process remember that it has sent a message. When a process terminates, if its is black, it sends a black token to its parent.

6. A black process turns back to white after it has sent a black token to its parent.
7. A parent process holding a black token (from one of its children), sends only a black token to its parent, to indicate that a message-passing was involved in its subtree.
8. Tokens are propagated to the root in this fashion. The root, upon receiving a black token, will know that a process in the tree had sent a message to some other process. Hence, it restarts the algorithm by sending a Repeat signal to all its children.
9. Each child of the root propagates the Repeat signal to each of its children and so on, until the signal reaches the leaves.
10. The leaf nodes restart the algorithm on receiving the Repeat signal.
11. The root concludes that termination has occurred, if:

    (a) it is white;
    (b) it is idle; and
    (c) it has received a white token from each of its children.

3. A spanning-tree-based termination detection algorithm

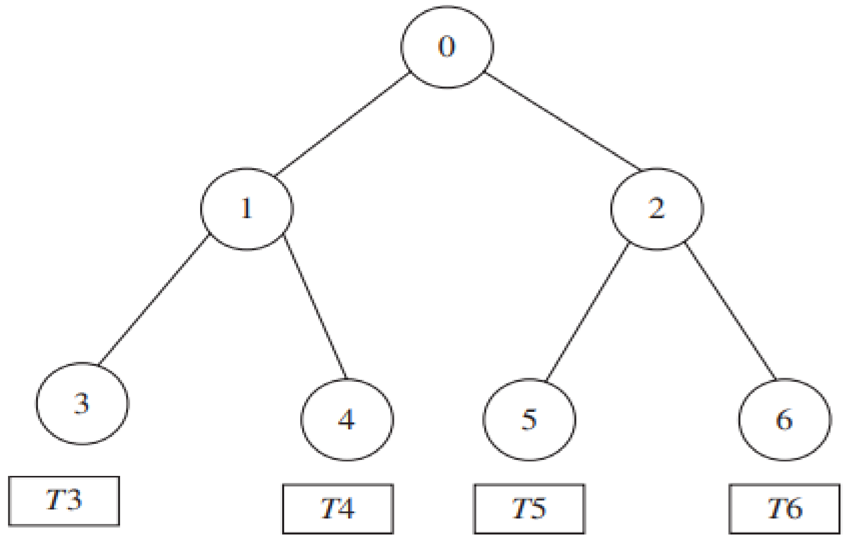


Figure 7.2 All leaf nodes have tokens. S = {3, 4, 5, 6}.

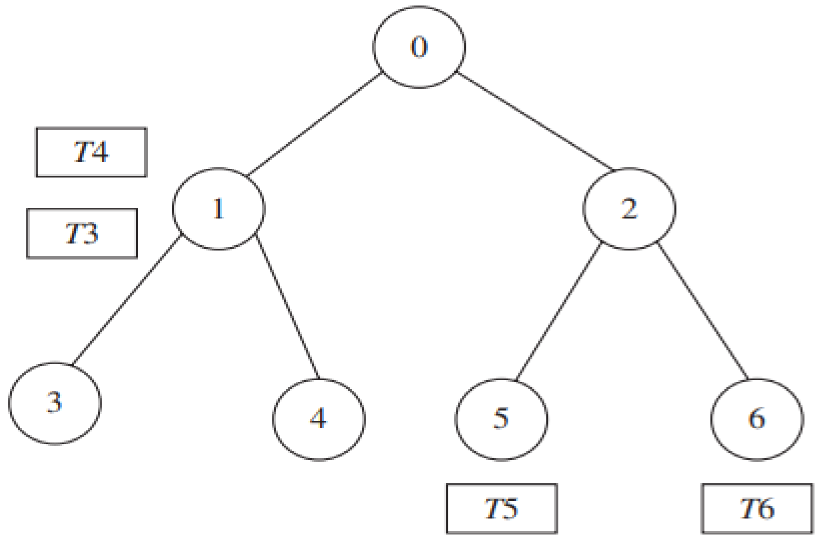


Figure 7.3 Nodes 3 and 4 become idle. *S* = {1, 5, 6}.

3. A spanning-tree-based termination detection algorithm

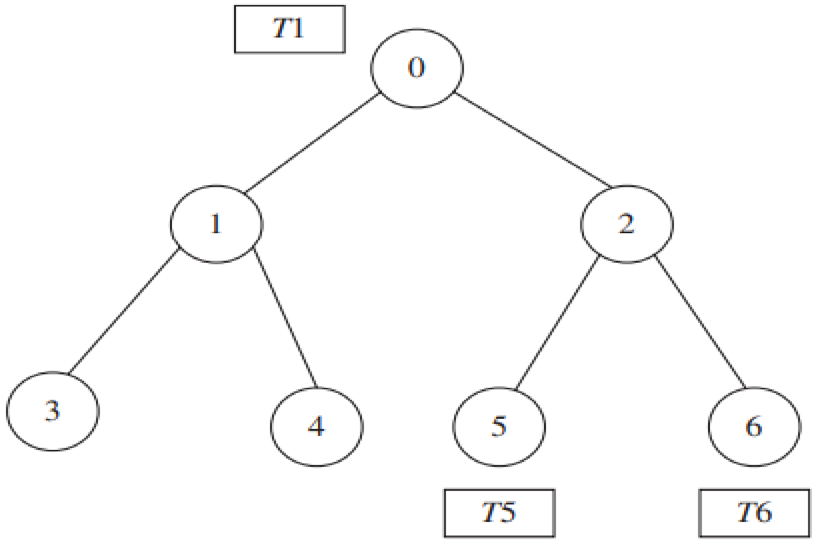


Figure 7.4 Node 1 becomes idle. $S = \{0, 5, 6\}$.

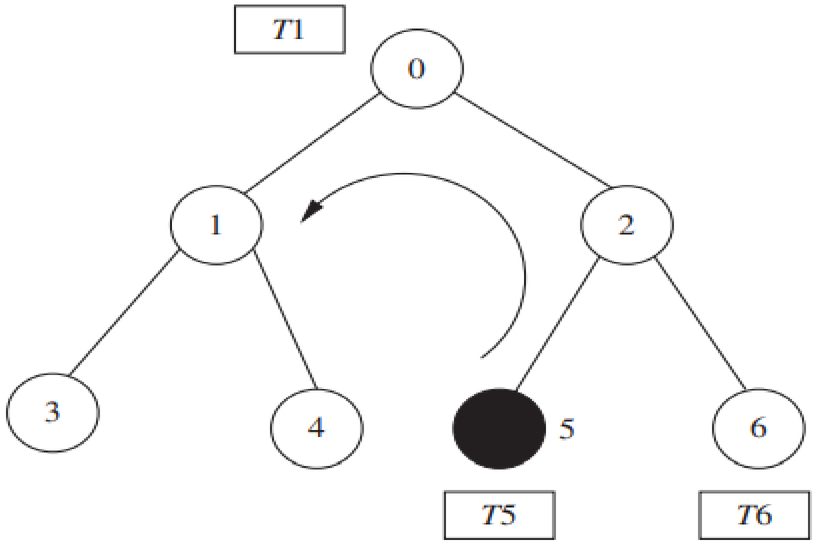


Figure 7.5 Node 5 sends a message to node 1.

3. A spanning-tree-based termination detection algorithm

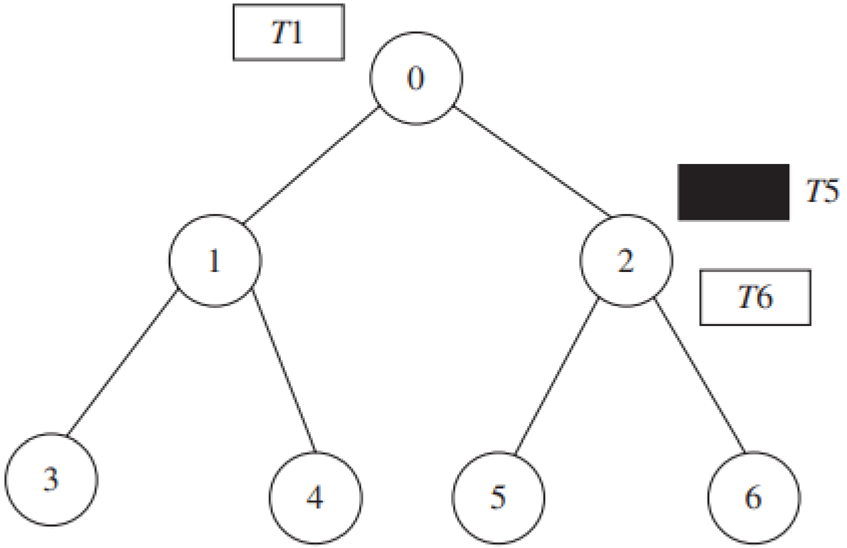


Figure 7.6 Nodes 5 and 6 become idle. $S = \{0, 2\}$.

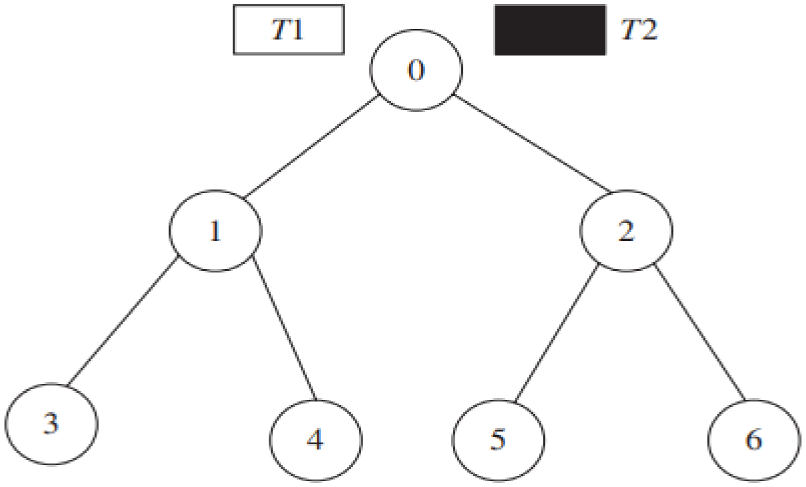
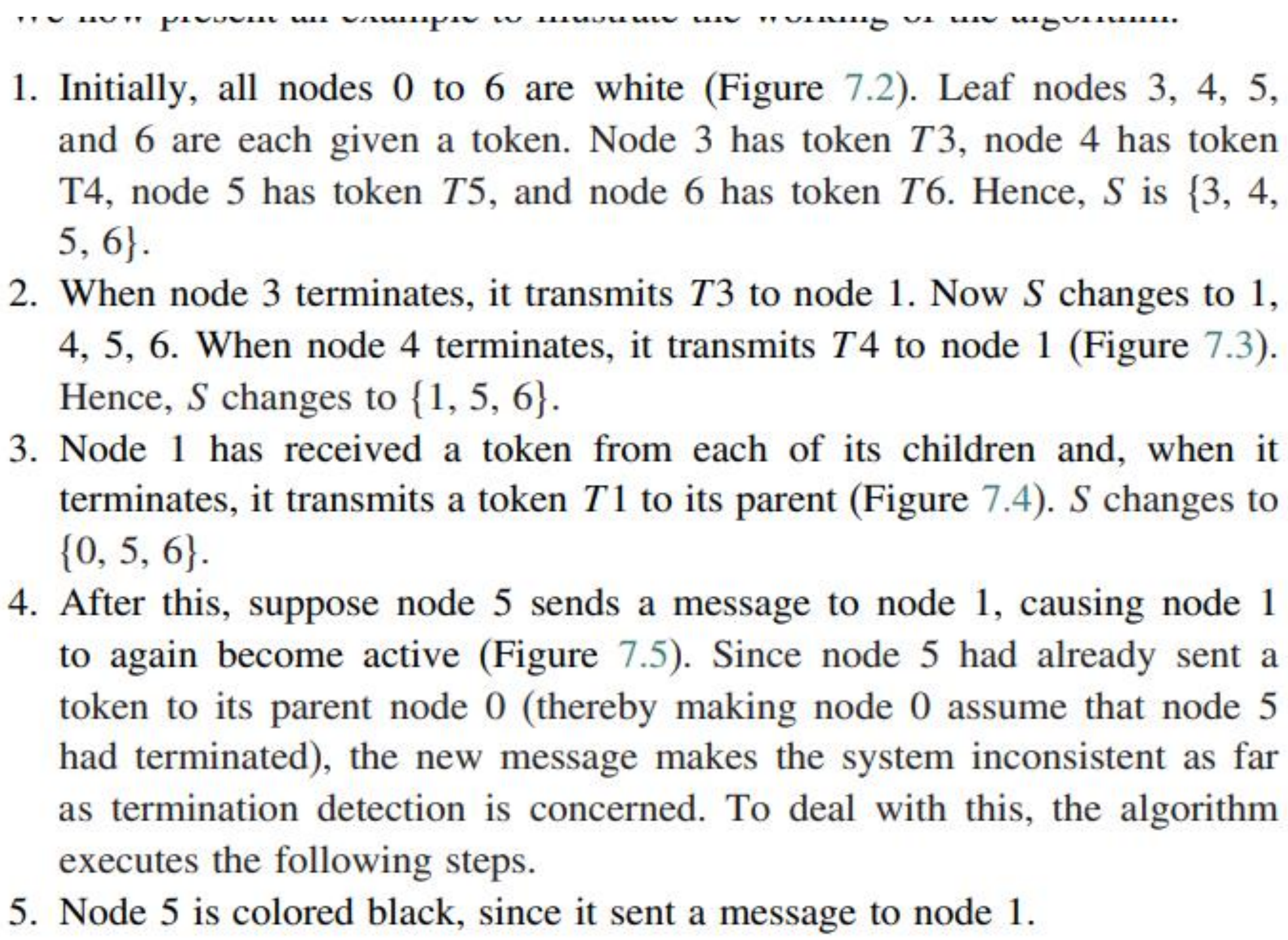


Figure 7.7 Node 2 becomes idle. $S = \{0\}$. Node 0 initiates a repeat signal.

3. A spanning-tree-based termination detection algorithm



We now present an example to illustrate the working of the algorithm.

1. Initially, all nodes 0 to 6 are white (Figure 7.2). Leaf nodes 3, 4, 5, and 6 are each given a token. Node 3 has token $T3$, node 4 has token T4, node 5 has token $T5$, and node 6 has token $T6$. Hence, S is {3, 4, 5, 6}.
2. When node 3 terminates, it transmits $T3$ to node 1. Now S changes to 1, 4, 5, 6. When node 4 terminates, it transmits $T4$ to node 1 (Figure 7.3). Hence, S changes to {1, 5, 6}.
3. Node 1 has received a token from each of its children and, when it terminates, it transmits a token $T1$ to its parent (Figure 7.4). S changes to {0, 5, 6}.
4. After this, suppose node 5 sends a message to node 1, causing node 1 to again become active (Figure 7.5). Since node 5 had already sent a token to its parent node 0 (thereby making node 0 assume that node 5 had terminated), the new message makes the system inconsistent as far as termination detection is concerned. To deal with this, the algorithm executes the following steps.
5. Node 5 is colored black, since it sent a message to node 1.

3. A spanning-tree-based termination detection algorithm

6. When node 5 terminates, it sends a black token *T*5 to node 2. So, *S* changes to {0, 2, 6}. After node 5 sends its token, it turns white (Figure 7.6). When node 6 terminates, it sends the white token *T*6 to node 2. Hence, *S* changes to {0, 2}.
7. When node 2 terminates, it sends a black token *T*2 to node 0, since it holds a black token *T*5 from node 5 (Figure 7.7).

8. Since node 0 has received a black token *T*2 from node 2, it knows that there was a message sent by one or more of its children in the tree and hence sends a repeat signal to each of its children.
9. The repeat signal is propagated to the leaf nodes and the algorithm is repeated. Node 0 concludes that termination has occurred if it is white, it is idle, and it has received a white token from each of its children.