DISTRIBUTED COMPUTING
MODULE-2

# Introduction

Usually causality is tracked using physical time. However, in distributed systems, it is not possible to have global physical time; it is possible to realize only an approximation of it. The knowledge of the causal precedence relation among the events of processes helps solve a variety of problems in distributed systems. Examples of some of these problems is as follows:

Distributed algorithms design The knowledge of the causal precedence relation among events helps ensure liveness and fairness in mutual exclusion algorithms, helps maintain consistency in replicated databases, and helps design correct deadlock detection algorithms to avoid phantom and undetected deadlocks.

Tracking of dependent events In distributed debugging, the knowledge of the causal dependency among events helps construct a consistent state for resuming reexecution; in failure recovery, it helps build a checkpoint; in replicated databases, it aids in the detection of file inconsistencies in case of a network partitioning.

Knowledge about the progress The knowledge of the causal dependency among events helps measure the progress of processes in the distributed computation. This is useful in discarding obsolete information, garbage collection, and termination detection.

Concurrency measure The knowledge of how many events are causally dependent is useful in measuring the amount of concurrency in a computation. All events that are not causally related can be executed concurrently. Thus, an analysis of the causality in a computation gives an idea of the concurrency in the program.

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 <. 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:

$$C : H \rightarrow T,$$

such that the following property is satisfied:

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

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

for two events e_i 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*.

# Implementation of 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 p_i maintains data structures that allow it the following two capabilities:

- A *local logical clock*, denoted by lc_i, that helps process pi 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

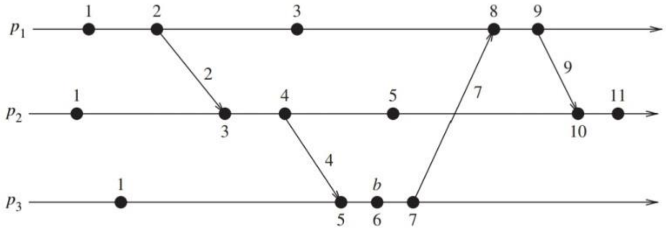
The protocol ensures that a process's logical clock, and thus its view of the global time, is managed consistently. The protocol consists of the following two rules:

- **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

- Proposed by Lamport in 1978 as an attempt to totally order events in a distributed system.
- Time domain 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 C_i .
- Rules R1 and R2 to update the clocks are as follows:R1: Before executing an event (send, receive, or internal), process pi executes the following: $C_i := C_i + d$ $(d > 0)$
- In general, every time R1 is executed, d can have a different value; however, typically d is kept at 1.
- R2: Each message piggybacks the clock value of its sender at sending time. When a process pi receives a message with timestamp C_{msg} , it executes the following actions:
  □ $C_i := \max(C_i, C_{msg})$
  □ Execute R1.
  □ Deliver the message.
  Figure shows evolution of scalar time



# Basic properties of scalar time

Consistency Property

- Scalar clocks satisfy the monotonicity and hence the consistency property:
  for two events ei and e_j , $e_i \rightarrow e_j \Longrightarrow C(e_i) < C(e_j)$.

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, the third event of process P1 and the second event of process P2 have identical scalar timestamp
- 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: $x \prec y \Leftrightarrow (h < k \text{ or } (h = k \text{ and } i < j))$

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, in Figure, 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(ei) < C(ej) $\not\Rightarrow$ ei → ej .
- For example, in Figure, 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.
- For example, in Figure, when process P2 receives the first message from process P1, it updates its clock to 3, forgetting that the timestamp of the latest event at P1 on which it depends is 2.

# 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 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 pj local time.
- If $vt_i[j]$=x, then process pi 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.
- Process pi uses the following two rules R1 and R2 to update its clock: R1: Before executing an event, process pi updates its local logical time as follows: vt_i [i] := vt_i [i] + d ; (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 pi executes the following sequence of actions:
    1. Update its global logical time as follows: $1 \le k \le n : vt_i[k] := \max(vt_i[k], vt[k])$
    2. Execute R1.
    3. Deliver the message m

# Basic properties if 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 → y ⇔ vh < vk
  x || y ⇔ vh || 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 i th component of vector clock at process pi , vt_i [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, ∑vh[j] − 1 represents the total number of events that causally precede e in the distributed computation

# Election Algorithm

- In order to perform coordination, distributed systems employ the concept of coordinators.
- An algorithm for choosing a unique process to play a particular role(coordinator) is called an election algorithm.

- Election algorithm assumes that every active process in the system has a unique priority number.
- The process with highest priority will be chosen as the coordinator.
- When a coordinator fails, the algorithm elects that active process which has highest priority number.
- Then this number is send to every active process in the distributed system.

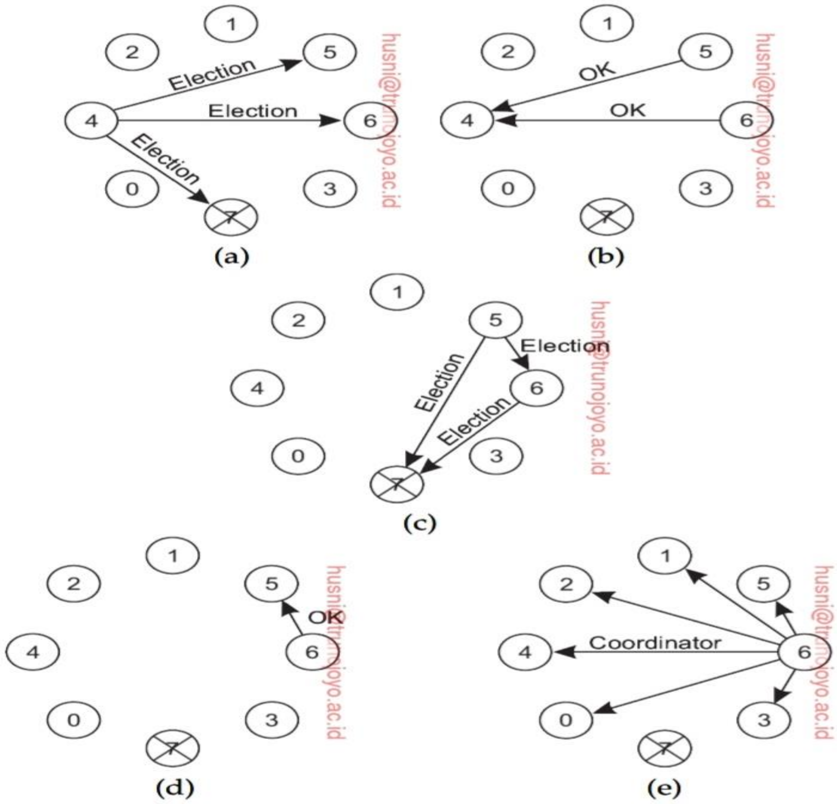
Bully Algorithm

There are 3 types of messages in bully algorithm

Election message – announces an election
Ok message – response to an election message
Coordinator message – announce the identity of the elected process

Steps:-

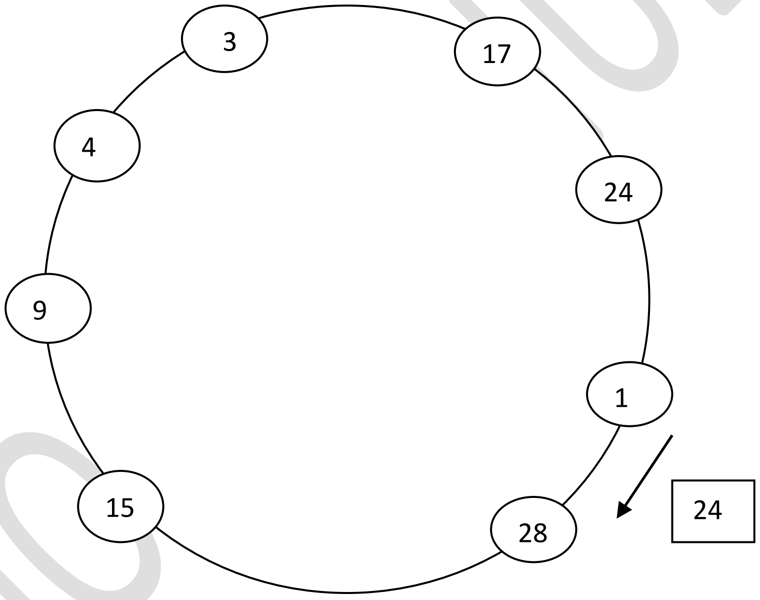
1. A process can begin an election by sending an election message to processes with high priority number and waiting for ok messages in response.
2. If none arrives within time T, the process considers itself as the coordinator and sends a coordinator message to all processes with lower identifiers announcing this
3. Otherwise the other process start election for a coordinator.
4. If coordinator does not respond to it within a time interval T, then it is assumed that coordinator has failed.
5. Now process P sends election message to every process with high priority number.
6. It waits for responses, if no one responds within time interval T, then process P elects itself as a coordinator.
7. Then it sends a message to all lower priority number processes that it is elected as their new coordinator.
8. If a process that was previously down/failed comes back it take over the coordinator job.
9. Biggest guy always wins hence the name bully algorithm.



Ring based election algorithm

- This algorithm applies to systems organized as a logical ring.

- In this algorithm we assume that the link between the processes are unidirectional.
- Every process can message to other process in clockwise direction only.
- Initially every process is marked as a non participant in an election.
- Any process can begin an election.
- It proceeds by making 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, then it substitutes its own identifier in the message and forwards it.
- 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 coordinator and sends an elected message to its neighbour.



- The election wasstarted by process 17.
- Process forward to neighbour with greatest identifier
- The election message currently contains 24, and forwards
- The process 28 will replace 24 with its identifier when the message reaches it
- The election message currently contains 28, and
- Forwards until the received identifier is that of the receiver itself,
- It becomes the coordinator and sends a coordinator message to its neighbours

# 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.

System model

- The system consists of a collection of n processes p_1, p_2, ..., p_n that are connected by channels.
- There are no globally shared memory and physical global clock and processes communicate by passing messages through communication channels.
- C_{ij} denotes the channel from process p_i to process p_j and its state is denoted by SC_{ij} .
- The actions performed by a process are modeled as three types of events: Internal events, the message send event and the message receive event.
- 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.
- 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 p_i till that instant.
- For an event e and a process state LS_i , $e \in LS_i$ iff e belongs to the sequence of events that have taken process p_i to state LS_i .
- For an event e and a process state LS_i , $e \notin LS_i$ iff e does not belong to the sequence of events that have taken process p_i to state LS_i .
- For a channel C_{ij} , the following set of messages can be defined based on the local states of the processes p_i and p_j

  Transit: $\text{transit}(LS_i , LS_j) = \{m_{ij} \mid \text{send}(m_{ij}) \in LS_i \land \text{rec}(m_{ij}) \notin LS_j \}$
- Thus, if a snapshot recording algorithm records the state of processes p_i and p_j as LS_i and LS_j , respectively, then it must record the state of channel C_{ij} as transit ($LS_i\_LS_j$).
- There are several models of communication among processes and different snapshot algorithms have assumed different models of communication.
- In FIFO model, each channel acts as a first-in first-out message queue and thus, message ordering is preserved by a channel.
- In non-FIFO model, a channel acts like a set in which the sender process adds messages and the receiver process removes messages from it in a random order.
- A system that supports causal delivery of messages satisfies the following property: "For any two messages m_{ij} and m_{kj} , if send(m_{ij}) −→ send(m_{kj}), then rec(m_{ij}) −→ rec(m_{kj})".

Consistent global state

The global state of a distributed system is a collection of the local states of the processes and the channels. Notationally, global state GS is defined as, $GS = \{ U_i LS_i , U_{i,j} SC_{ij} \}$
A global state GS is a consistent global state iff it satisfies the following two conditions :

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

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

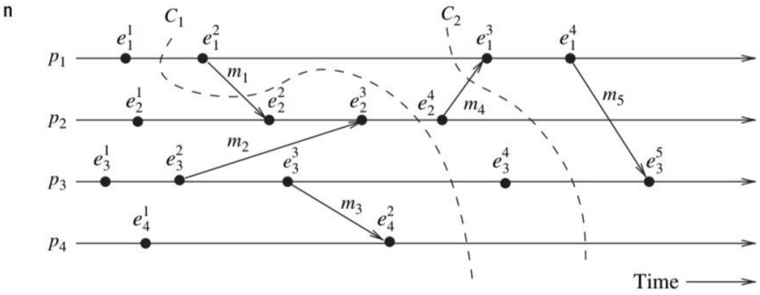
Condition **C1** states the law of conservation of messages. Every message m_{ij} that is recorded as sent in the local state of a process p_i must be captured in the state of the channel C_{ij} or in the collected local state of the receiver process p_j .

Condition **C2** states that in the collected global state, for every effect, its cause must be present. If a message m_{ij} is not recorded as sent in the local state of process pi, then it must neither be present in the state of the channel C_{ij} nor in the collected local state of the receiver process pj . In a consistent global state, every message that is recorded as received is also recorded as sent.

Interpretation in terms of cuts

- A cut in a space-time diagram is a line joining an arbitrary point on each process line that slices the space-time diagram into a PAST and a FUTURE.
- 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 was sent in the PAST of that cut. Such a cut is known as a consistent cut.
- For example, consider the space-time diagram for the computation illustrated in Figure. Cut C_1 is inconsistent because message m_1 is flowing from the FUTURE to the PAST. Cut C_2 is consistent and message m_4 must be captured in the state of channel C_{21}.

- Note that in a consistent snapshot, all the recorded local states of processes are concurrent; that is, the recorded local state of no process casually affects the recorded local state of any other process.



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.
- The snapshot of channel Cij includes all the messages that process pj receives after taking the snapshot and whose timestamp is smaller than the time of the snapshot. (All messages are timestamped with the sender's clock.) Clearly, if channels are not FIFO, a termination detection scheme will be needed to determine when to stop waiting for messages on channels.
- 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
- **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 pj must record its snapshot before processing a message mij that was sent by process pi after recording its snapshot.

# Snapshot algorithms for FIFO channels

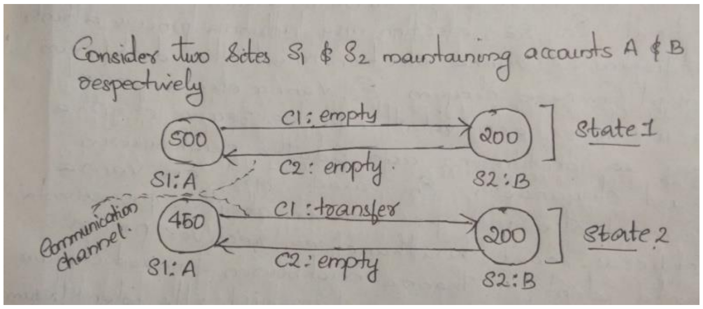
Global Snapshot = Global State = collection of individual local states of each process in the distributed system + individual state of each communication channel in the distributed system

Snapshot:- is a photograph of a process taken or recorded quickly

Need for taking snapshots or recording global state of a system

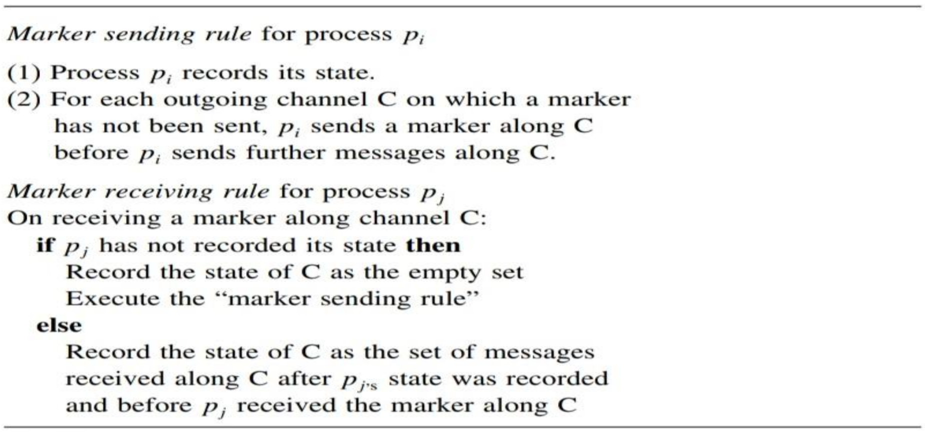
1. Check pointing:- snapshot will be used as a checkpoint, to restart the application in case of failure
2. Collecting garbage:- used to remove objects that don't have any references
3. Detecting deadlocks:- used to examine the current application state.
4. Termination detection

The reason we need to record the local state and communication channel state can be explained using an example:


Consider two sites S_1 & S_2 maintaining accounts A & B respectively

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.



```
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 and Complexity

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 p_j receives message m_{ij} that precedes the marker on channel C_{ij} , it acts as follows: If process p_j has not taken its snapshot yet, then it includes m_{ij} in its recorded snapshot. Otherwise, it records m_{ij} in the state of the channel C_{ij} . Thus, condition C1 is satisfied.

Complexity

- The recording part of a single instance of the algorithm requires O(e) messages and O(d) time, where e is the number of edges in the network and d is the diameter of the network.

Properties of the recorded global state

The recorded global state may not correspond to any of the global states that occurred during the computation.

This happens because a process can change its state asynchronously before the markers it sent are received by other sites and the other sites record their states.

 But the system could have passed through the recorded global states in some equivalent executions.

 The recorded global state is a valid state in an equivalent execution and if a stable property (i.e., a property that persists) holds in the system before the snapshot algorithm begins, it holds in the recorded global snapshot.

 Therefore, a recorded global state is useful in detecting stable properties

# Termination Detection

A fundamental problem in distributed systems is to determine if a distributed computation has terminated. The detection of the termination of a distributed computation is non-trivial since no process has complete knowledge of the global state, and global time does not exist. 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.

Messages used in the underlying computation are called *basic* messages, and messages used for the purpose of termination detection (by a termination detection algorithm) are called *control* messages.
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.

System model of a distributed computation

A distributed computation has the following characteristics:
At any given time, a process can be in only one of the two states: active, where it is doing local computation and idle, where the process has (temporarily) finished the execution of its local computation and will be reactivated only on the receipt of a message from another process.
An active process can become idle at any time.
An idle process can become active only on the receipt of a message from another process.
Only active processes can send messages.
A message can be received by a process when the process is in either of the two states, i.e., active or idle. On the receipt of a message, an idle process becomes active.
The sending of a message and the receipt of a message occur as atomic actions

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: $(\forall i :: p_i(t_0) = \textit{idle}) \wedge (\forall i, j :: c_{i,j}(t_0) = 0)$
Thus, a distributed computation has terminated iff all processes have become idle and there is no message in transit in any channel

Termination detection Using Distributed Snapshots

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.
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 before 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. If a request is successful,

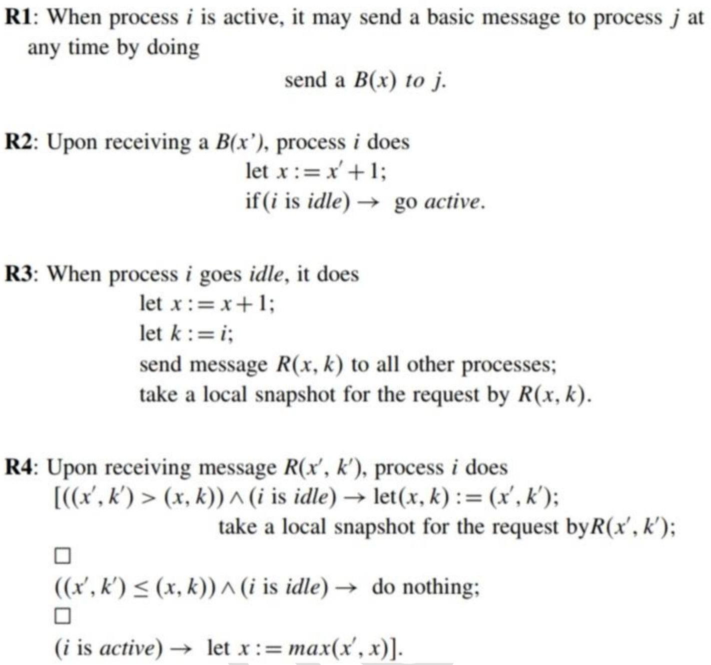
global snapshot of the request can thus be obtained and the recorded state will indicate termination of the computation

Formal description:

Each process *i* maintains a logical clock denoted by *x*, 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 its logical time *x* is of the form *B(x)*. A control message that requests processes to take local snapshot issued by process *i* at its logical time *x* is of the form *R(x, i)*. 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. A process also 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'

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.

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 $x := max(x', x)$].
```

The last process to terminate will have the largest clock value. Therefore, every process will take a snapshot for it, however, it will not take a snapshot for any other process.

# 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.

Basic Idea:

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. A non-zero weight W (0<W ≤ 1) is assigned to each process in the active state and to each message in transit in the following manner:

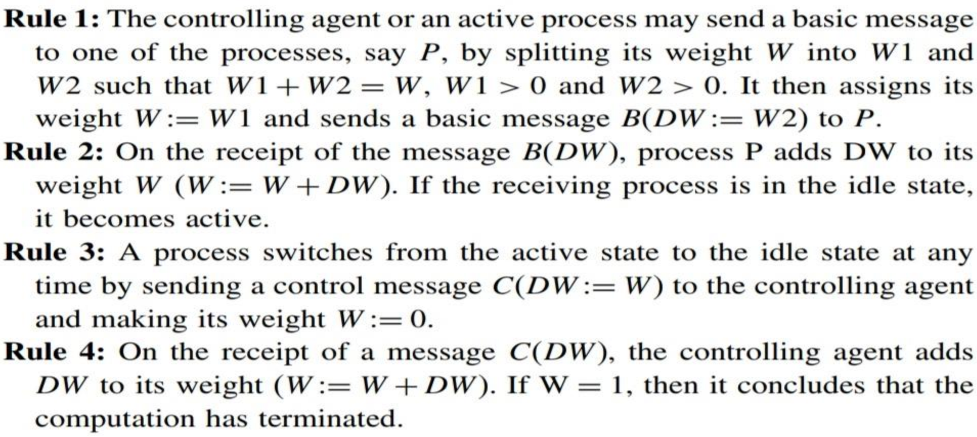
- When a process sends a message, it sends a part of its weight in the message.
- When a processreceives a message, it adds the weight received in the message to it's weight.
- Thus, the sum of weights on all the processes and on all the messages in transit is always 1.
- When a process becomes passive, 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.

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.

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.

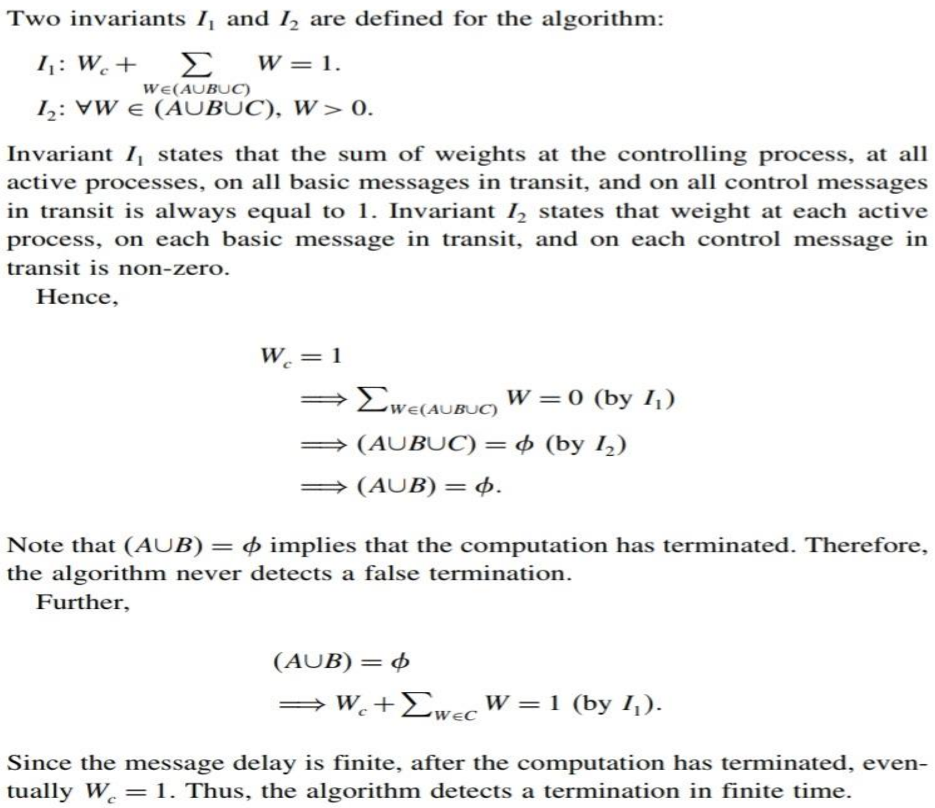
Correctness of Algorithm:

A: set of weights on all active processes

B: set of weights on all basic messages in transit

C: set of weights on all control messages in transit

W_c : weight on the controlling agent.



Two invariants I_1 and I_2 are defined for the algorithm:

$$I_1: W_c + \sum_{W\in(A\cup B\cup C)} W = 1.$$

$$I_2: \forall W \in (A\cup B\cup C),\ W > 0.$$

Invariant I_1 states that the sum of weights at the controlling process, at all active processes, on all basic messages in transit, and on all control messages in transit is always equal to 1. Invariant I_2 states that weight at each active process, on each basic message in transit, and on each control message in transit is non-zero.

Hence,

$$\begin{aligned} W_c &= 1 \\ &\Longrightarrow \textstyle\sum_{W\in(A\cup B\cup C)} W = 0 \text{ (by } I_1) \\ &\Longrightarrow (A\cup B\cup C) = \phi \text{ (by } I_2) \\ &\Longrightarrow (A\cup B) = \phi. \end{aligned}$$

Note that $(A\cup B) = \phi$ implies that the computation has terminated. Therefore, the algorithm never detects a false termination.

Further,

$$\begin{aligned} &(A\cup B) = \phi \\ &\Longrightarrow W_c + \textstyle\sum_{W\in C} W = 1 \text{ (by } I_1). \end{aligned}$$

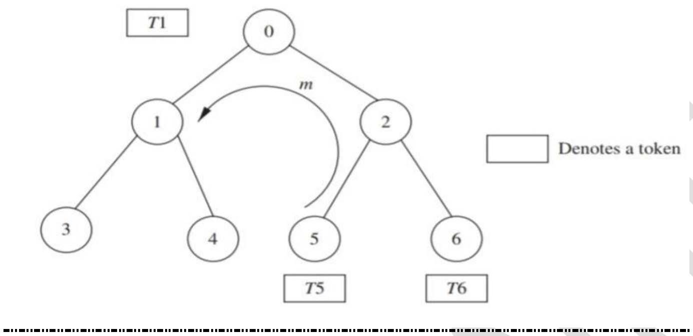
Since the message delay is finite, after the computation has terminated, eventually $W_c = 1$. Thus, the algorithm detects a termination in finite time.

# Spanning-Tree-Based Termination Detection Algorithm

- There are N processes Pi , 0≤i≤N, which are modeled as the nodes i, 0≤i≤N, of a fixed connected undirected graph.
- The edges of the graph represent the communication channels.
- 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 through 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.
- Two waves of signals generated one moving inward and other 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.
- 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, when a process after it has sent a token to its parent, receives a message from some other process, which could cause the process to again become active (See Figure).
- Hence the simple algorithm fails since the process that indicated to its parent that it has become idle, is now active because of the message it received from an active process.
- Hence, 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.



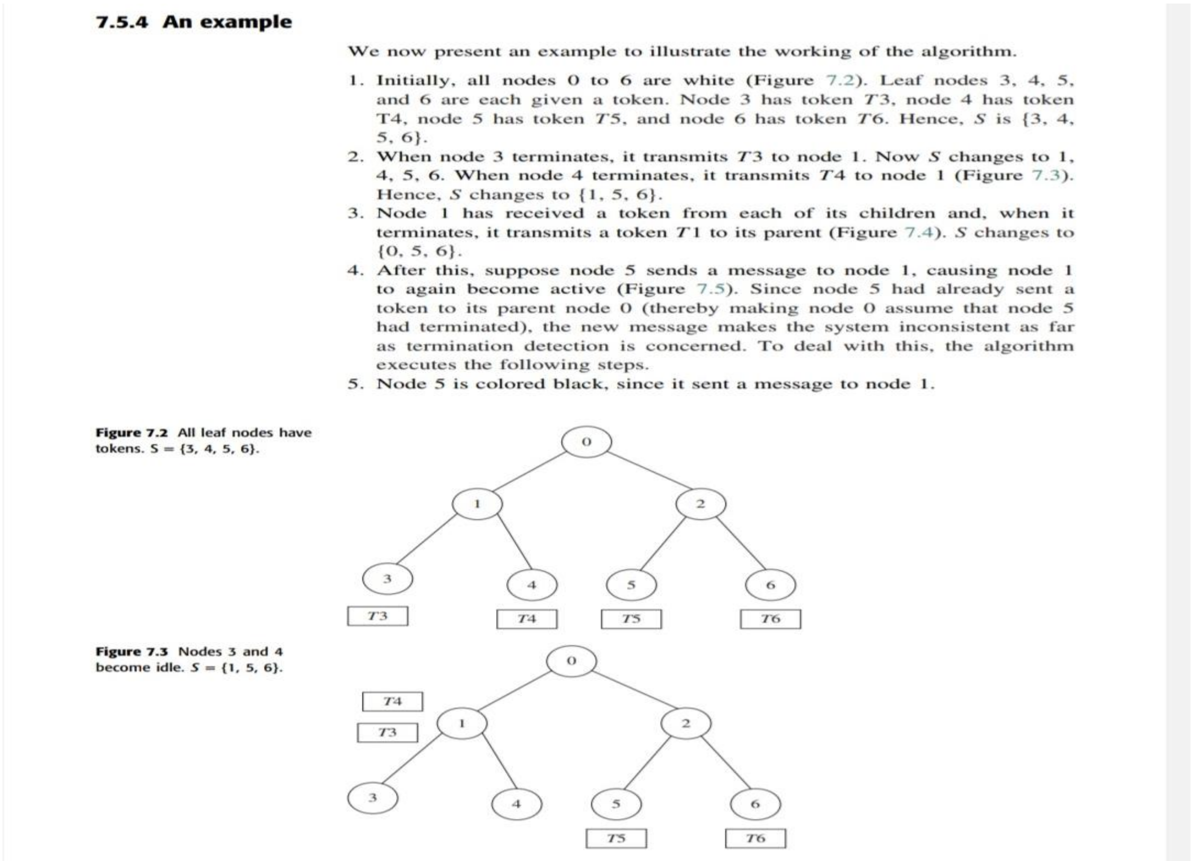
Main idea is to color the processes and tokens and change the color when messages such as in Figure are involved.

The algorithm works as follows:

- 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.
- Initially, all processes and tokens are colored white.
- When a leaf node terminates, it sends the token it holds to its parent process.
- 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.
- A process turns black when it sends a message to some other process. When a process terminates, if its color is black, it sends a black token to its parent.
- A black process turns back to white, after it has sent a black token to its parent.
- 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.
- 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.
- Each child of the root propagates the Repeat signal to each of its children and so on, until the signal reaches the leaves.
- The leaf nodes restart the algorithm on receiving the Repeat signal.
- The root concludes that termination has occurred, if it is white, it is idle, and it received a white token from each of its children.

Performance

- The best case message complexity of the algorithm is O(N), where N is the number of processes in the computation, which occurs when all nodes send all computation messages in the first round.
- The worst case complexity of the algorithm is O(N*M), where M is the number of computation messages exchanged, which occurs when only computation message is exchanged every time the algorithm is executed.



7.5.4 An example

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 *T*3, node 4 has token T4, node 5 has token *T*5, and node 6 has token *T*6. Hence, *S* is {3, 4, 5, 6}.
2. When node 3 terminates, it transmits *T*3 to node 1. Now *S* changes to 1, 4, 5, 6. When node 4 terminates, it transmits *T*4 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 *T*1 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.

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}.

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

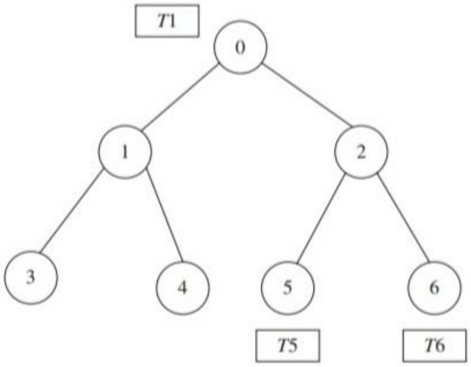


Figure 7.5 Node 5 sends a message to node 1.

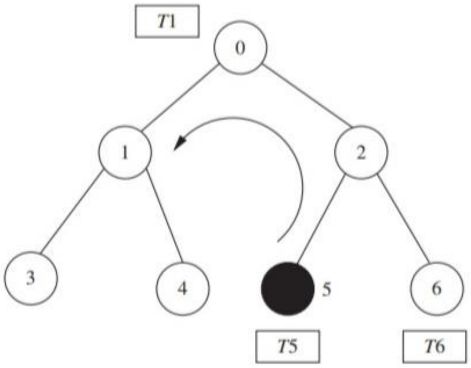
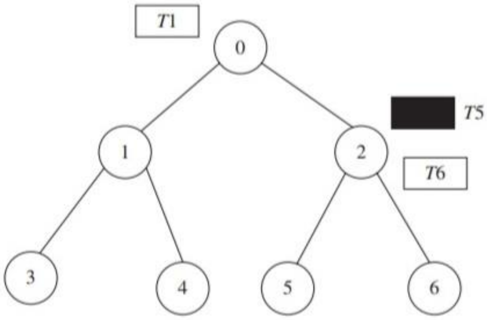
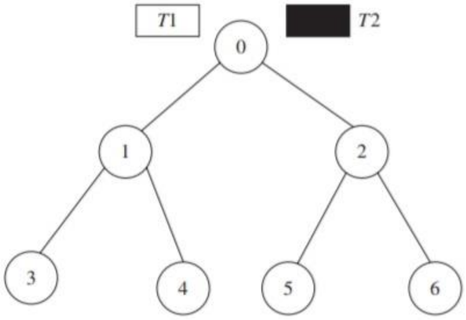


Figure 7.6 Nodes 5 and 6 become idle. S = {0, 2}.



6. When node 5 terminates, it sends a black token $T5$ 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 $T6$ to node 2. Hence, S changes to {0, 2}.
7. When node 2 terminates, it sends a black token $T2$ to node 0, since it holds a black token $T5$ from node 5 (Figure 7.7).

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



8. Since node 0 has received a black token $T2$ 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.