RAMAIAH
Institute of Technology

M.S. Ramaiah Institute of Technology
(Autonomous Institute, Affiliated to VTU)
Department of Computer Science and Engineering

Course Name: Distributed Systems

Course Code – CS751

Credits - 3:0:0

UNIT -2

Term: JULY 2023 – NOV 2023

Prepared by: Dr. Sangeetha. V
Associate Professor

RAMAIAH Institute of Technology

Textbooks

Textbook:

1. Ajay D. Kshemkalyani, and Mukesh Singhal "**Distributed Computing: Principles, Algorithms, and Systems**", Cambridge University Press, 2008 (Reprint 2013).

Reference Books:

1. John F. Buford, Heather Yu, and Eng K. Lua, "**P2P Networking and Applications**", Morgan Kaufmann, 2009 Elsevier Inc.
2. Kai Hwang, Geoffrey C. Fox, and Jack J. Dongarra, "**Distributed and Cloud Computing: From Parallel processing to the Internet of Things**", Morgan Kaufmann, 2012 Elsevier Inc.

RAMAIAH
Institute of Technology

Other CIE Component(20 marks)

NPTEL: Distributed Systems, IIT Patna

Dr. Rajiv Misra

https://nptel.ac.in/courses/106106168

Course Duration : Jul-Sep 2023- This a **8 Weeks** course will begin on **July 24, 2023**

Enrollment :2023-05-08 to 2023-07-31

Last date for exam registration: Aug 18, 2023, 5:00 PM (Friday). Exam fees will be **Rs. 1000/- per exam**.

Date of exam: September 24, 2023

RAMAIAH
Institute of Technology

UNIT 2

Global state and snapshot recording algorithms: Introduction, System model and definitions, Snapshot algorithms for FIFO channels, Variations of the Chandy–Lamport algorithm, Snapshot algorithms for non-FIFO channels, Snapshots in a causal delivery system, Monitoring global state.

Terminology and basic algorithms: Topology abstraction and overlays, Classifications and basic concepts, Synchronizers, Maximal independent set (MIS), Leader election. Complexity measures and metrics, Program structure, Elementary graph algorithms.

RAMAIAH
Institute of Technology

UNIT 2

Global state and snapshot recording algorithms: (Chapter 4 -4.1 to 4.7)

- Introduction,
- System model and definitions,
- Snapshot algorithms for FIFO channels,
- Variations of the Chandy–Lamport algorithm,
- Snapshot algorithms for non-FIFO channels,
- Snapshots in a causal delivery system,
- Monitoring global state.

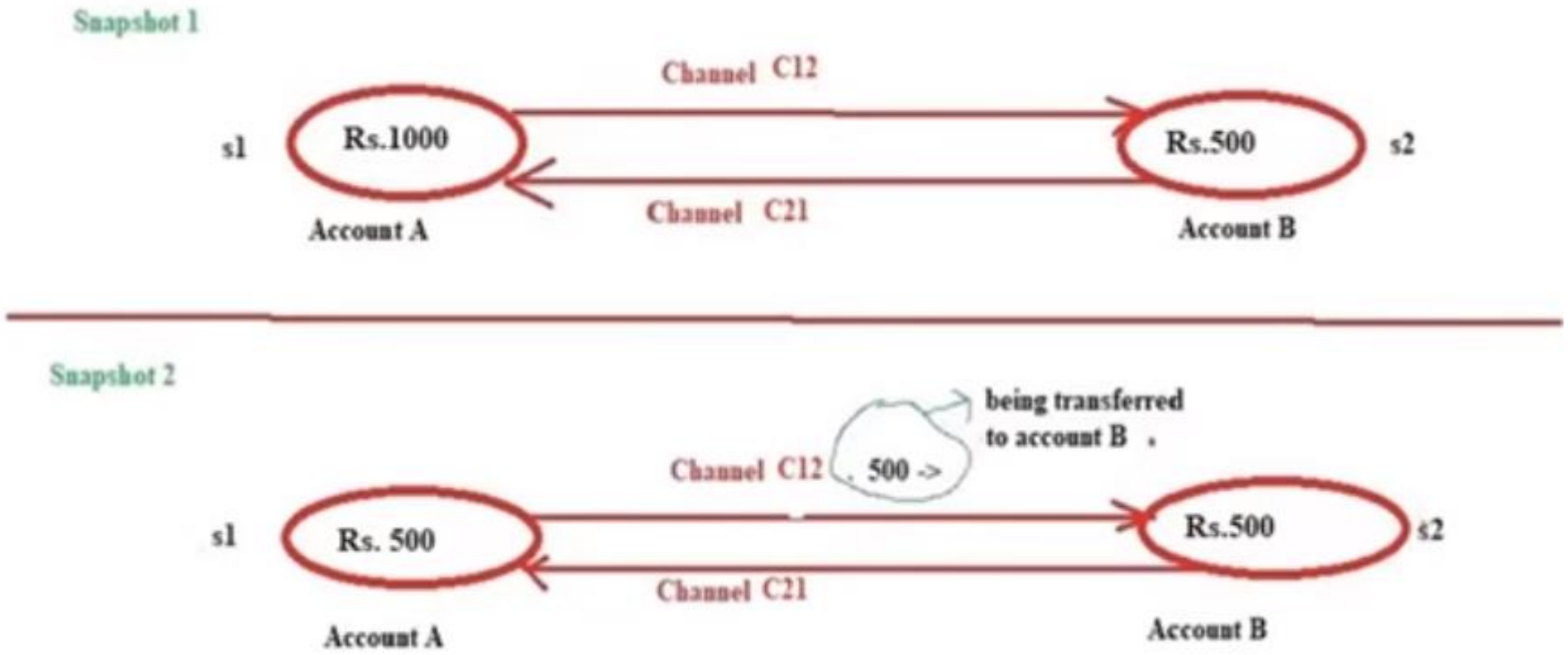
Introduction

- 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.

Introduction

- Recording the global state of a distributed system on-the-fly is an important paradigm.
- The lack of globally shared memory, global clock and unpredictable message delays in a distributed system make this problem non-trivial.
- This chapter first defines consistent global states and discusses issues to be addressed to compute consistent distributed snapshots.
- Then several algorithms to determine on-the-fly such snapshots are presented for several types of networks.

Introduction



System model and definitions

- 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.

System model and definitions

- 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: $transit(LS_i, LS_j) = \{m_{ij} \mid send(m_{ij}) \in LS_i \bigwedge rec(m_{ij}) \notin LS_j \}$

System model and definitions

Recall, there are three models of communication: FIFO, non-FIFO, and Co.

- 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 $send(m_{ij}) \longrightarrow send(m_{kj})$, then $rec(m_{ij}) \longrightarrow rec(m_{kj})$".

System model and definitions

Consistent global state

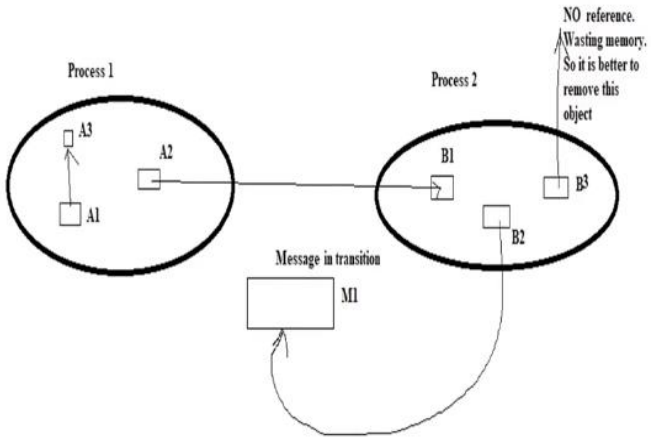
- The global state of a distributed system is a collection of the local states of the processes and the channels.
- Notationally, global state GS is defined as,

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

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



RAMAIAH Institute of Technology

System model and definitions

Check pointing:

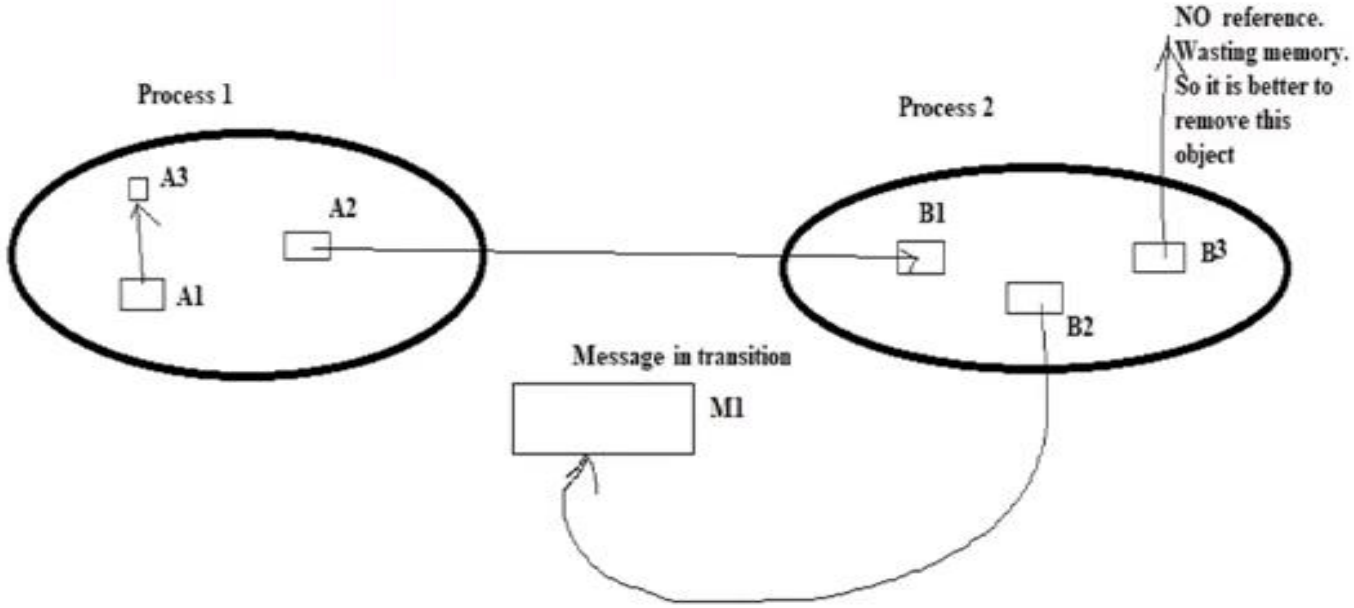
Snapshot will use as a checkpoint , to restart the application in case of failure

Collecting garbage:

Use to remove objects that don't have any references

Detecting deadlocks:

Use to examine the current application state



System model and definitions

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 FUTUR

RAMAIAH
Institute of Technology

System model and definitions

Interpretation in terms of cuts

- 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.

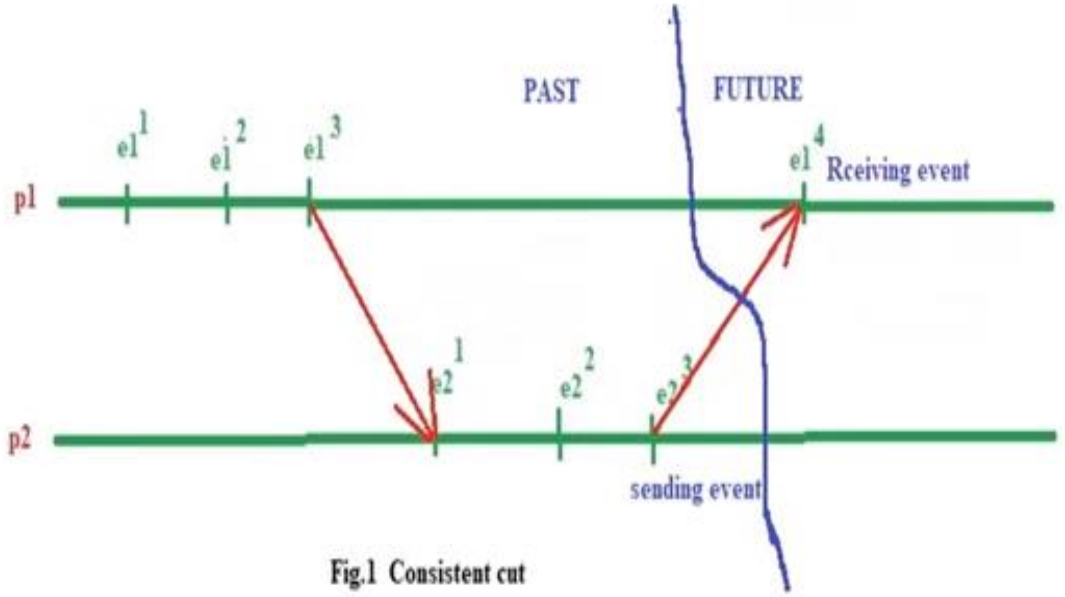


Fig.1 Consistent cut

System model and definitions

Interpretation in terms of cuts

- Inconsistent cut.

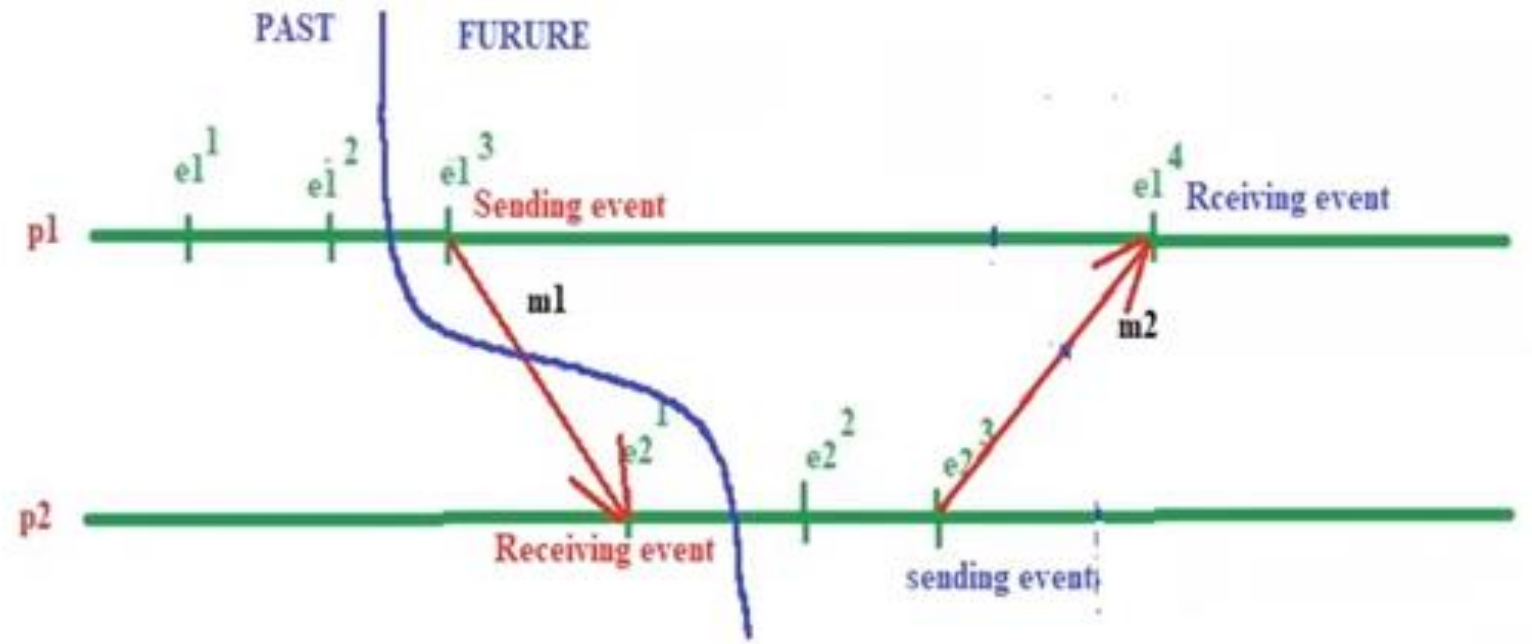
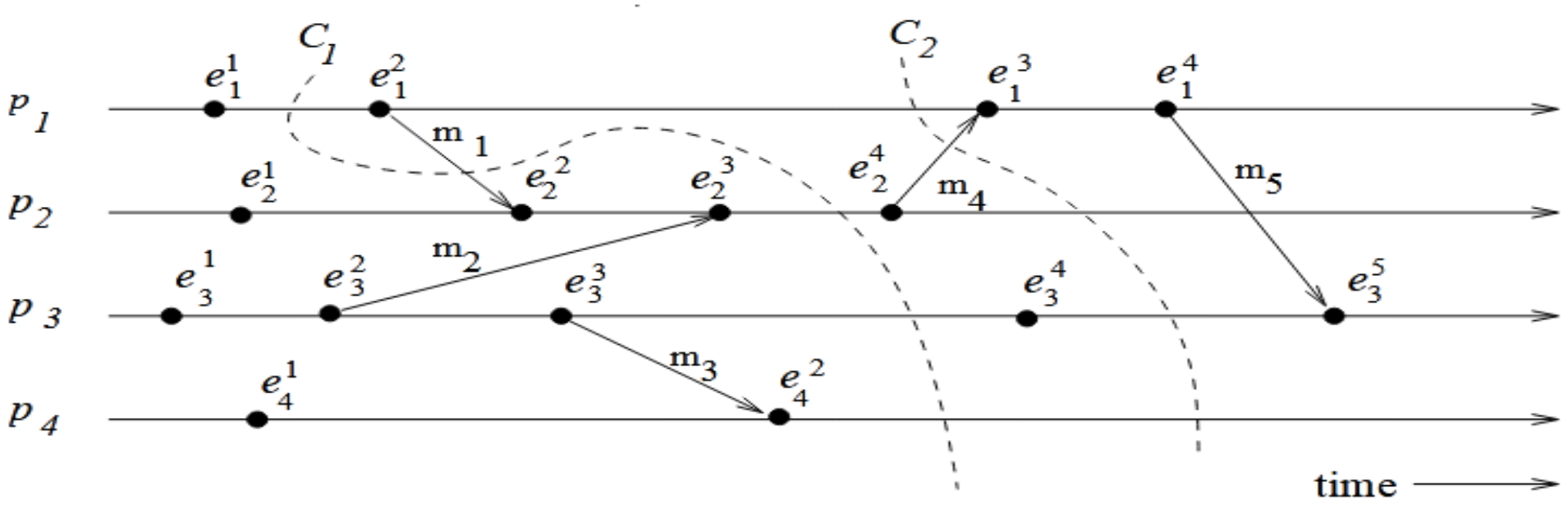


Fig 2. Inconsistent cut

System model and definitions



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 C_{21}.

System model and definitions

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

1. Chandy-Lamport algorithm

Variations of the Chandy–Lamport algorithm

1. Spezialetti–Kearns algorithm
2. Venkatesan's incremental snapshot algorithm
3. Helary's wave synchronization method

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.

Chandy-Lamport algorithm

- 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.

Chandy-Lamport algorithm

Marker Sending Rule for process *i*

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

Marker Receiving Rule for process *j*

On receiving a marker along channel C:

if *j* has not recorded its state **then**

Record the state of C as the empty set

Follow the "Marker Sending Rule"

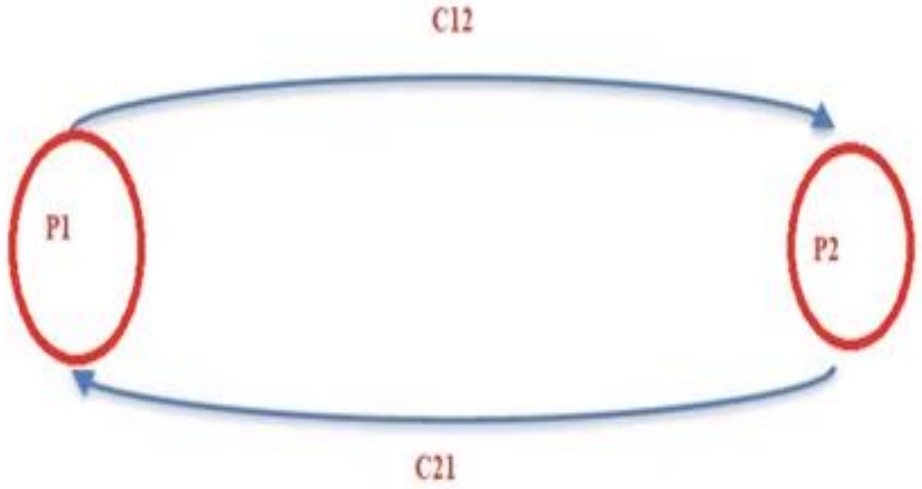
else

Record the state of C as the set of messages received along C after *j*'s state was recorded and before *j* received the marker along C

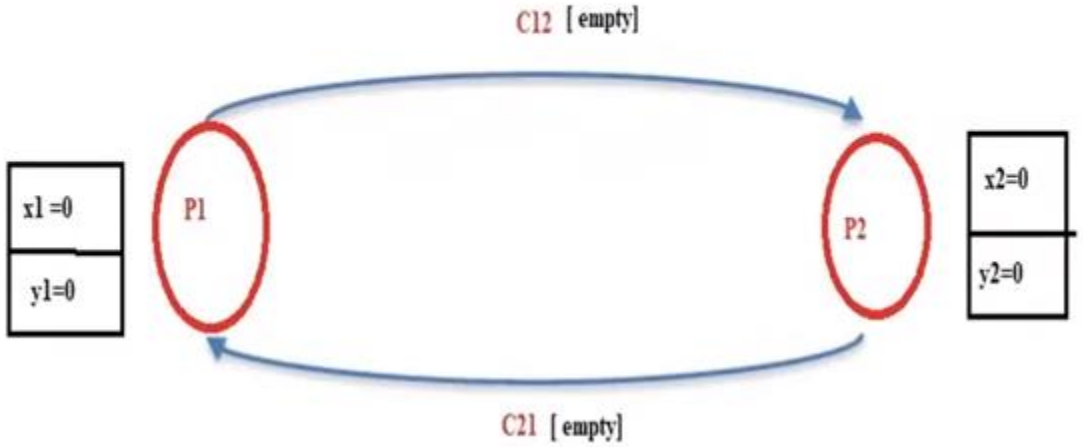
Chandy-Lamport algorithm

Let us assume,

two processes P1 & P2 are running
two channels are C12, C21



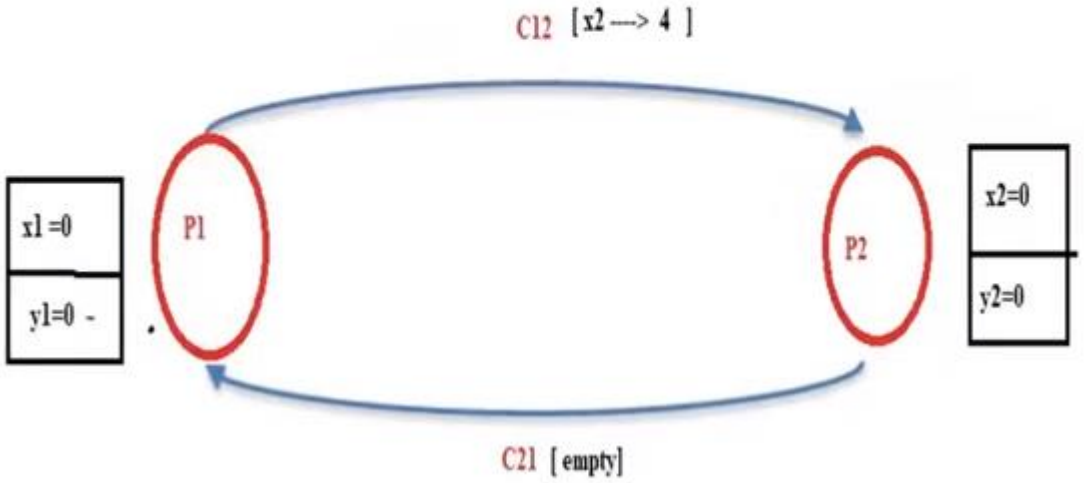
Initial global state (snapshot 1) of the processes and channels



Chandy-Lamport algorithm

Now, P1 tells P2 to change its state variable x2 to 4

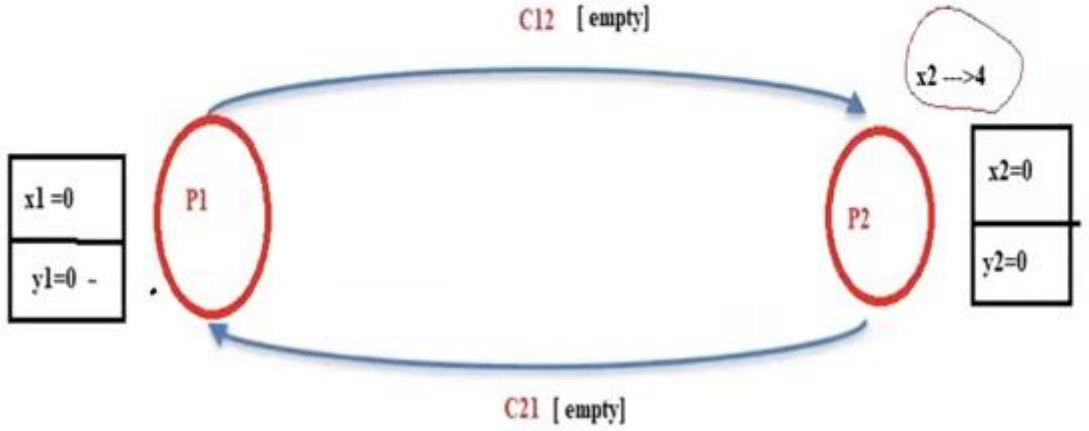
This is an another global snapshot of this distributed system – No: 2



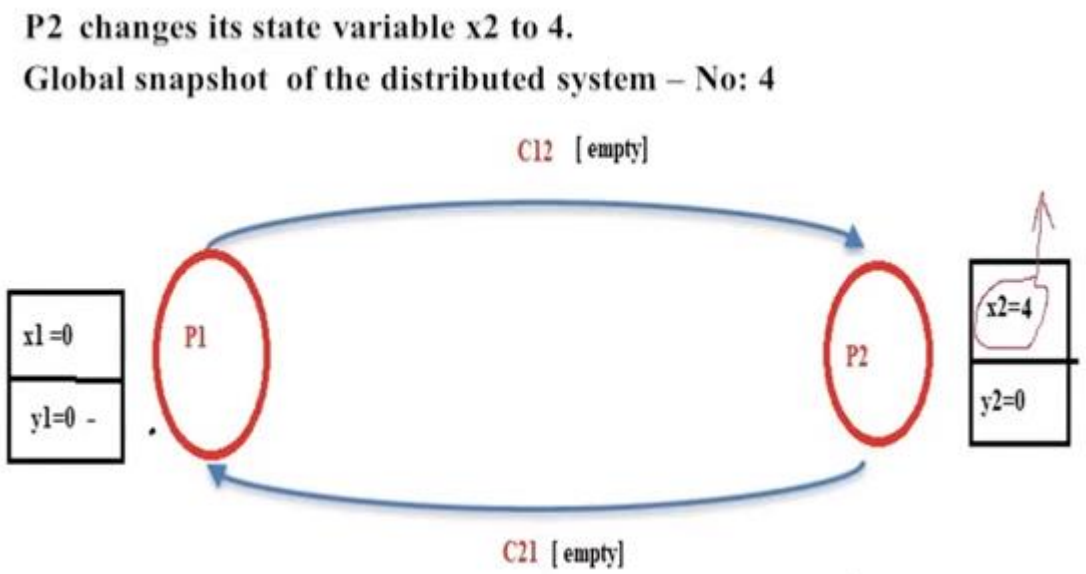
Now P2 receives the message sent by p1 through the channel C12.

. So the channel C_{12} becomes empty now.

Global snapshot of this distributed system - No: 3



Chandy-Lamport algorithm



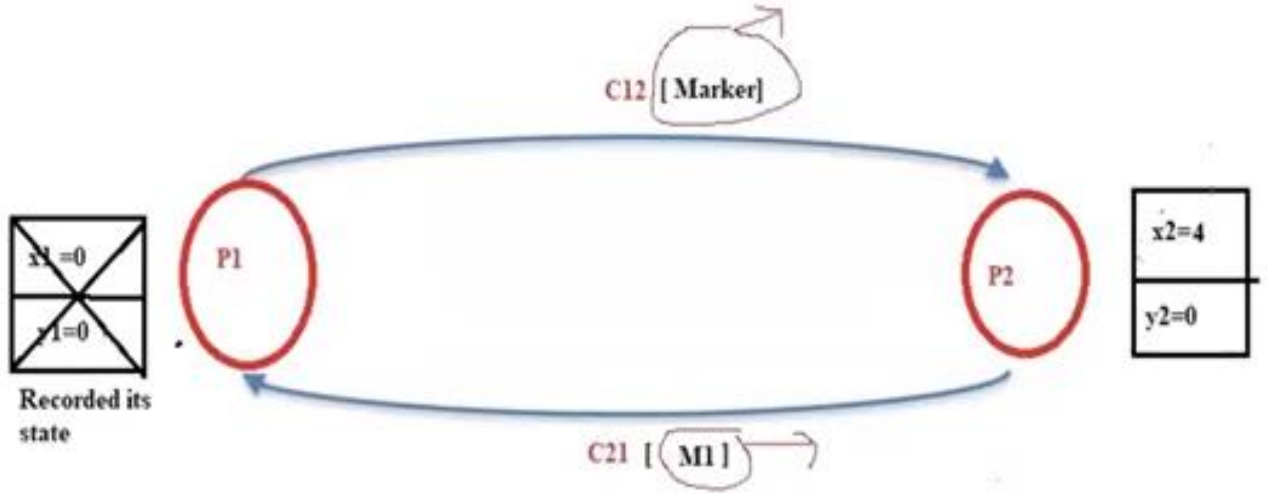
RAMAIAH Institute of Technology

Chandy-Lamport algorithm

Now , the Process P1 initiates the recording process. i.e P1 is the initiator .

So, P1 sends a MARKER message to P2 and begins recording its local state and recording all messages on inbound channels

Meanwhile, P2 also sent a message to P1



Marker Sending Rule for process *i*

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

Marker Receiving Rule for process *j*

On receiving a marker along channel C:

if *j* has not recorded its state **then**

Record the state of C as the empty set

Follow the "Marker Sending Rule"

else

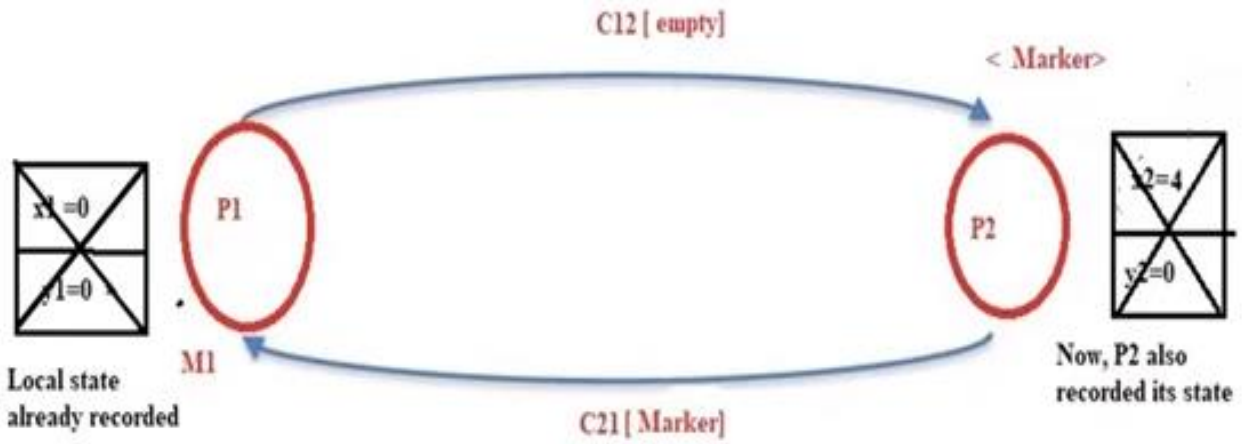
Record the state of C as the set of messages received along C after *j*'s state was recorded and before *j* received the marker along C

Chandy-Lamport algorithm

P2 receives the marker message for the first time, so records its local state.

Meanwhile P1 also receives the message M1 , sent by p2

Now, P2 sends a MARKER message to P1



Marker Sending Rule for process i

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

Marker Receiving Rule for process j

On receiving a marker along channel C:

if j has not recorded its state **then**

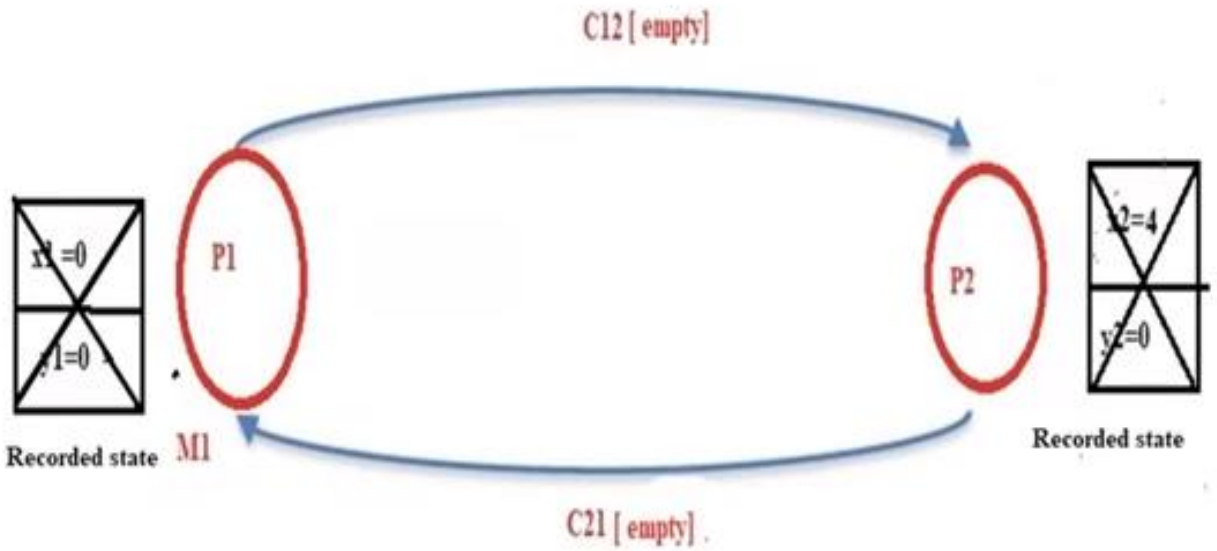
Record the state of C as the empty set
Follow the "Marker Sending Rule"

else

Record the state of C as the set of messages received along C after j's state was recorded and before j received the marker along C

Chandy-Lamport algorithm

Now the statue is, P1 has already sent a MARKER message to P2, so it records all messages it received on inbound channels to the appropriate channel's state



Marker Sending Rule for process *i*

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

Marker Receiving Rule for process *j*

On receiving a marker along channel C:

if *j* has not recorded its state **then**

Record the state of C as the empty set

Follow the "Marker Sending Rule"

else

Record the state of C as the set of messages received along C after *j*'s state was recorded and before *j* received the marker along C

Chandy-Lamport algorithm

Correctness

To prove the correctness of the algorithm, we show that a recorded snapshot satisfies conditions **C1 and C2.**

Since a process records its snapshot when it receives the first marker on any incoming channel, no messages that follow markers on the channels incoming to it are recorded in the process's snapshot.

Moreover, a process stops recording the state of an incoming channel when a marker is received on that channel.

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 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.

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

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

Chandy-Lamport algorithm

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.

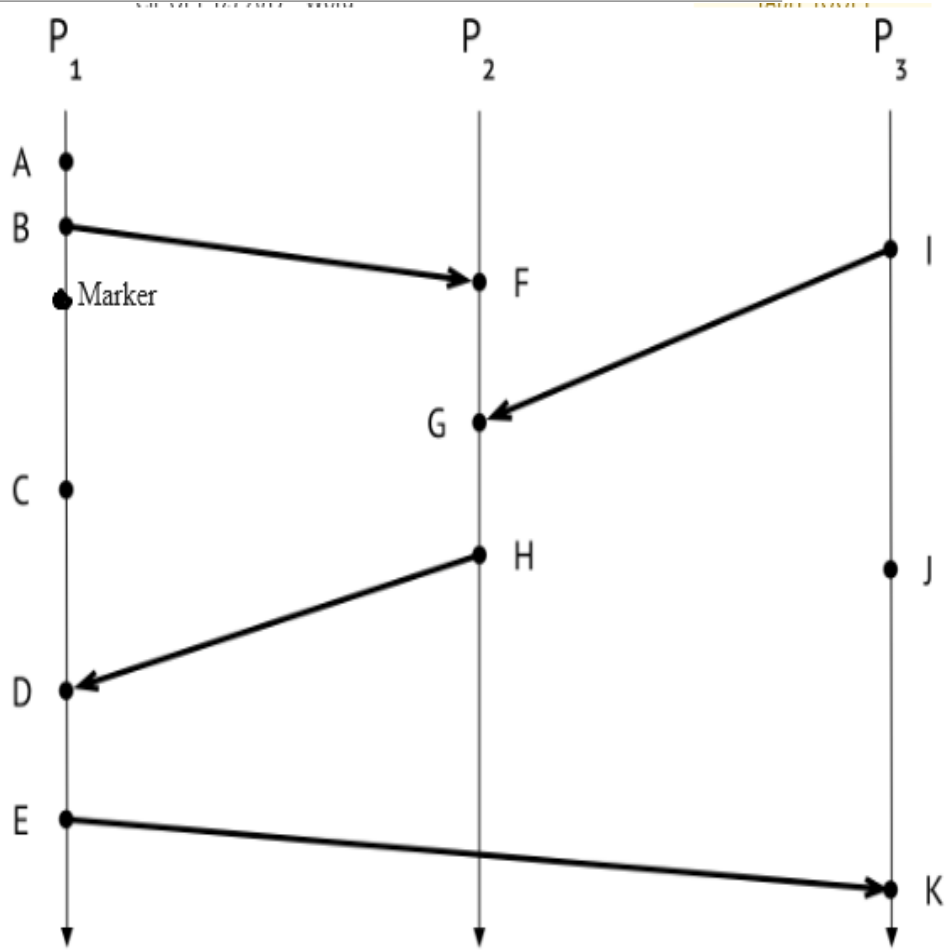
RAMAIAH Institute of Technology

Example

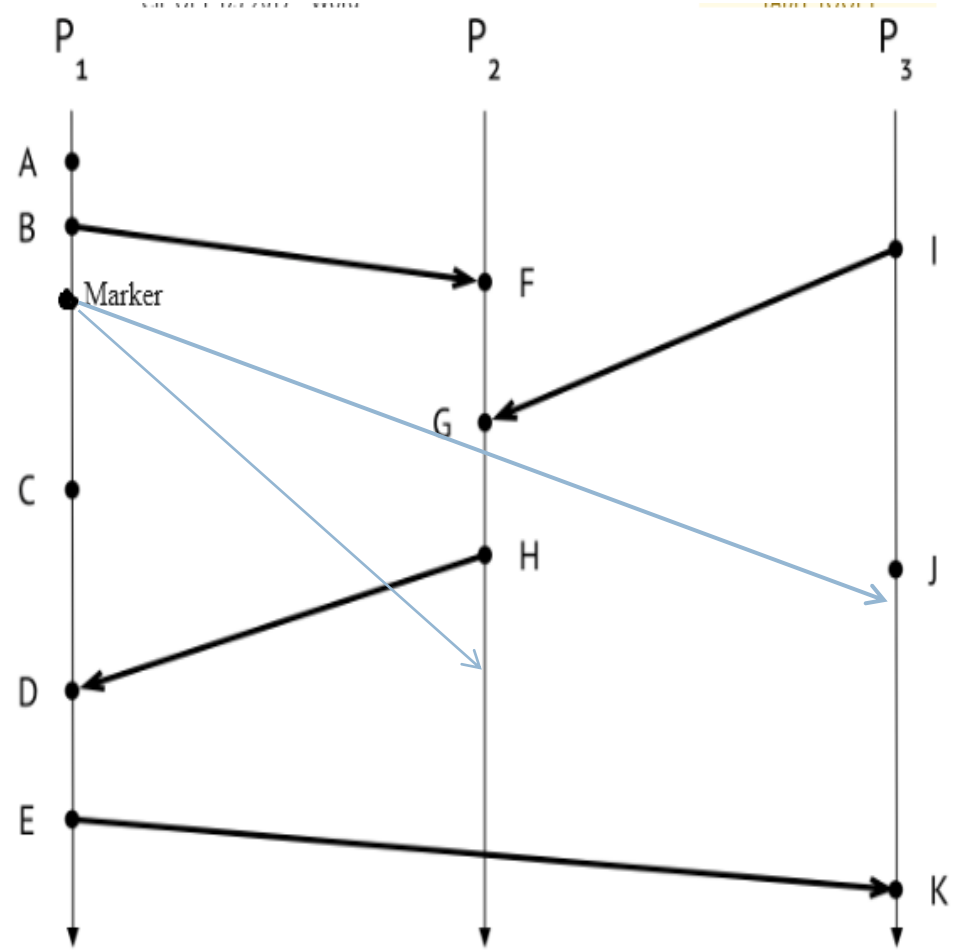
Apply Chandy Lamport Global Snapshot algorithm for the given setup with three processes, each with several events, including messages sent and received.

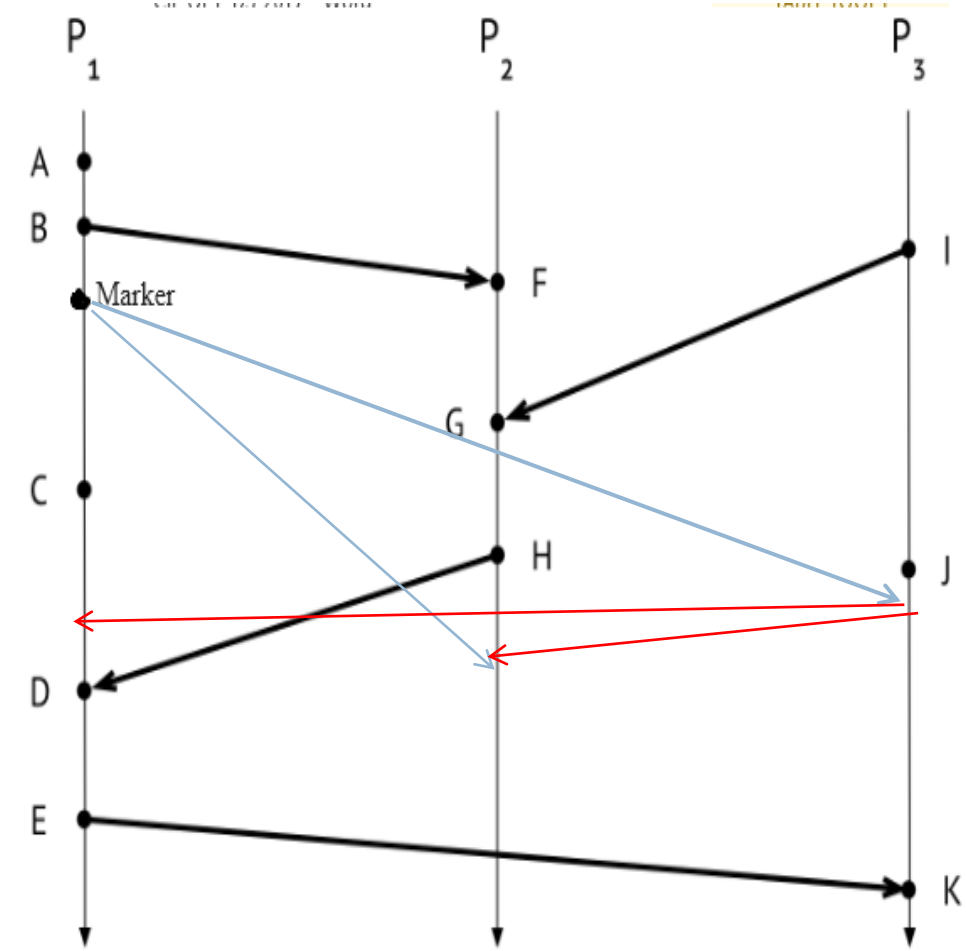
Illustrate the process.

A Marker is issued from P1 through all its communication channels. If the Local Snapshot is recorded by P2 after event H and P3 after event J, indicate whether the cuts in distributed execution is consistent or not.

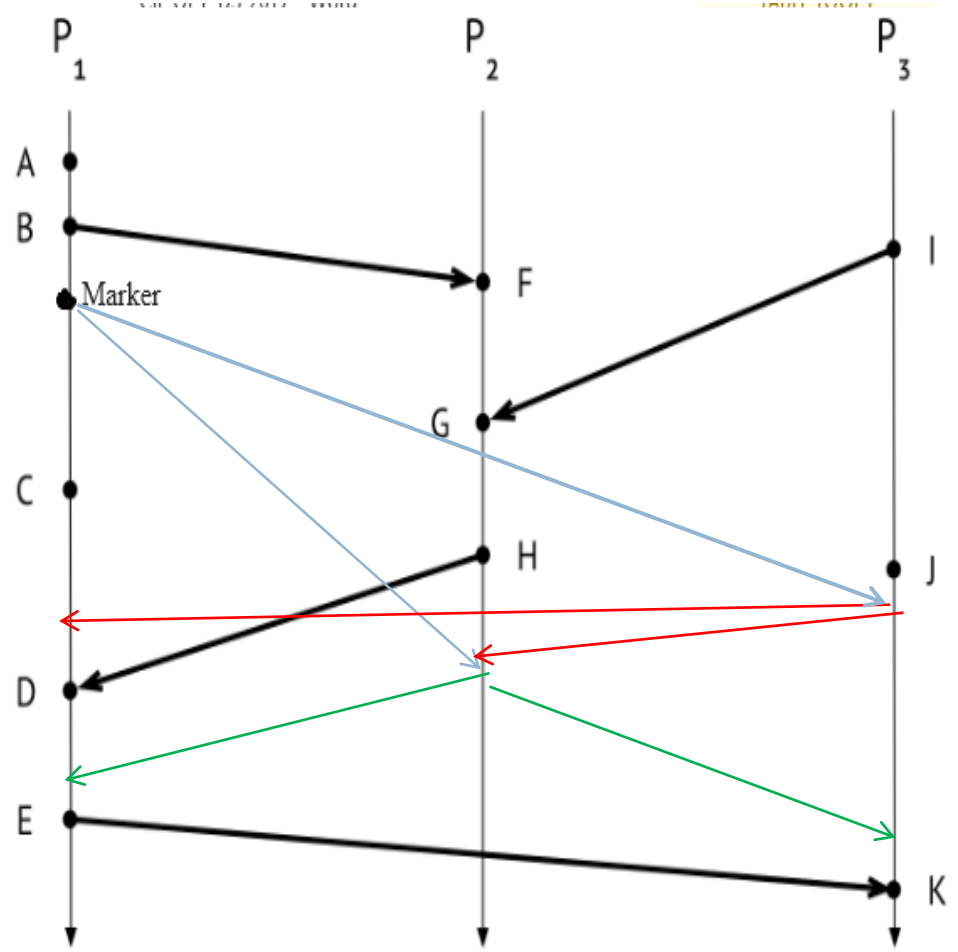


Example





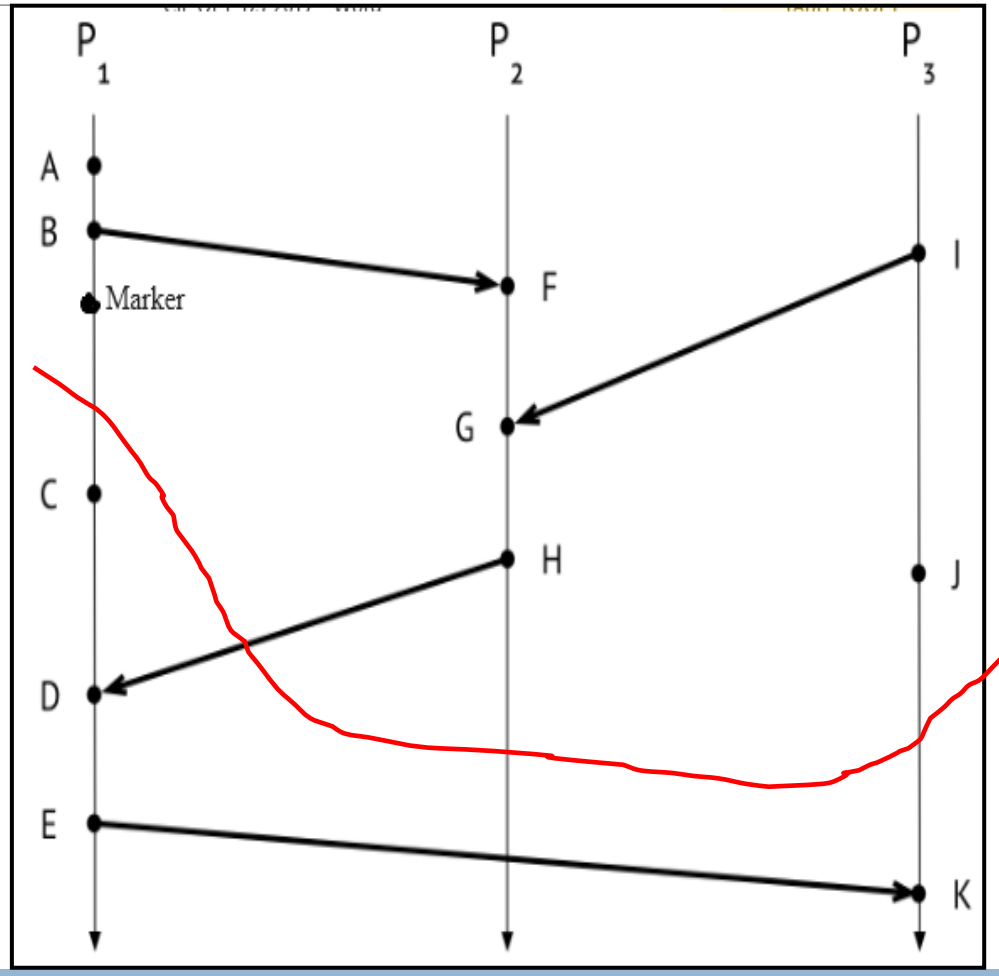
Example



Example

The cuts in distributed execution is consistent.

- Sending of H→ D is recorded and its message is saved in channel
- E→ K is completely future which satisfies condition C2(Both rec and send not captured)



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

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

Snapshot algorithms for FIFO channels

1. Chandy-Lamport algorithm

Variations of the Chandy–Lamport algorithm

1. Spezialetti–Kearns algorithm
2. Venkatesan's incremental snapshot algorithm
3. Helary's wave synchronization method

Snapshot algorithms for FIFO channels

Spezialetti–Kearns algorithm

There are two phases in obtaining a global snapshot:

1. **Efficient snapshot recording-**locally recording the snapshot at every process
2. **Efficient dissemination of the recorded snapshot-**distributing the resultant global snapshot to all the initiators.

Snapshot algorithms for FIFO channels

Spezialetti and Kearns provided two optimizations to the Chandy–Lamport algorithm.

1. The first optimization combines snapshots concurrently initiated by multiple processes into a single snapshot.
2. The second optimization deals with the efficient distribution of the global snapshot.

- A process needs to take only one snapshot, irrespective of the number of concurrent initiators and all processes are not sent the global snapshot.
- This algorithm assumes bi-directional channels in the system.

Snapshot algorithms for FIFO channels

Efficient snapshot recording

- A marker carries the identifier of the initiator of the algorithm.
- Each process has a variable master to keep track of the initiator of the algorithm.

RAMAIAH Institute of Technology

Snapshot algorithms for FIFO channels

Efficient snapshot recording

- A key notion used by the optimizations is that of a region in the system.
- A region encompasses all the processes whose master field contains the identifier of the same initiator.
- A region is identified by the initiator's identifier.
- When there are multiple concurrent initiators, the system gets partitioned into multiple regions.
- The identifier of the concurrent initiator is recorded in a local variable id-border-set.

Snapshot algorithms for FIFO channels

Efficient snapshot recording

- When a process executes the “marker sending rule” on the receipt of its first marker, it records the initiator’s identifier carried in the received marker in the master variable.
- A process that initiates the algorithm records its own identifier in the master variable.

Snapshot algorithms for FIFO channels

Efficient snapshot recording

- When the initiator's identifier in a marker received along a channel is different from the value in the master variable, a concurrent initiation of the algorithm is detected and the sender of the marker lies in a different region.
- The variable id-border-set at a process contains the identifiers of the neighboring regions.
- The process receiving the marker does not take a snapshot for this marker and does not propagate this marker.

Snapshot algorithms for FIFO channels

Efficient snapshot recording

- Thus, the algorithm efficiently handles concurrent snapshot initiations by suppressing redundant snapshot collections – a process does not take a snapshot or propagate a snapshot request initiated by a process if it has already taken a snapshot in response to some other snapshot initiation.

Snapshot algorithms for FIFO channels

Efficient dissemination of the recorded snapshot

The Spezialetti–Kearns algorithm efficiently assembles the snapshot as follows:

- In the snapshot recording phase, a forest of spanning trees is implicitly created in the system.
- The initiator of the algorithm is the root of a spanning tree and all processes in its region belong to its spanning tree.

RAMAIAH Institute of Technology

Snapshot algorithms for FIFO channels

Efficient dissemination of the recorded snapshot

1.If process pi executed the “marker sending rule” because it received its first marker from process pj, then process pj is the parent of process pi in the spanning tree

2.After an intermediate process in a spanning tree has received the recorded states from all its child processes and has recorded the states of all incoming channels, it forwards its locally recorded state and the locally recorded states of all its descendent processes to its parent.

Snapshot algorithms for FIFO channels

Efficient dissemination of the recorded snapshot

3.When the initiator receives the locally recorded states of all its descendents from its children processes, it assembles the snapshot for all the processes in its region and the channels incident on these processes.

- 4.The initiator knows the identifiers of initiators in adjacent regions using id-border-set information it receives from processes in its region

Snapshot algorithms for FIFO channels

Efficient dissemination of the recorded snapshot

- The initiator exchanges the snapshot of its region with the initiators in adjacent regions in rounds.
- In each round, an initiator sends to initiators in adjacent regions, any new information obtained from the initiator in the adjacent region during the previous round of message exchange.
- A round is complete when an initiator receives information, or the blank message (signifying no new information will be forthcoming) from all initiators of adjacent regions from which it has not already received a blank message

Snapshot algorithms for FIFO channels

1. Chandy-Lamport algorithm

Variations of the Chandy–Lamport algorithm

1. Spezialetti–Kearns algorithm
2. Venkatesan's incremental snapshot algorithm
3. Helary's wave synchronization method

Venkatesan's incremental snapshot algorithm

Many applications require repeated collection of global snapshots of the system.

Venkatesan proposed the following efficient approach:

- This approach modifies the global snapshot algorithm of Chandy–Lamport to save on messages when computation messages are sent only on a few of the network channels, between the recording of two successive snapshot

Venkatesan's incremental snapshot algorithm

The incremental snapshot algorithm assumes

- bidirectional FIFO channels,
- The presence of a single initiator,
- A fixed spanning tree in the network
- Four types of control messages: init_snap, regular, and ack. init_snap, and snap_completed messages traverse the spanning tree edges.
- regular and ack messages, which serve to record the state of non-spanning edges, are not sent on those edges on which no computation message has been sent since the previous snapshot

RAMAIAH
Institute of Technology

Venkatesan's incremental snapshot algorithm

The algorithm works as follows:

- Snapshots are assigned version numbers and all algorithm messages carry this version number.
- The initiator notifies all the processes the version number of the new snapshot by sending init_snap messages along the spanning tree edges.
- A process follows the "marker sending rule" when it receives this notification or when it receives a regular message with a new version number

Venkatesan's incremental snapshot algorithm

The algorithm works as follows:

- The "marker sending rule" is modified so that the process sends regular messages along only those channels on which it has sent computation messages since the previous snapshot, and the process waits for ack messages in response to these regular messages.
- When a leaf process in the spanning tree receives all the ack messages it expects, it sends a snap_completed message to its parent process.

Venkatesan's incremental snapshot algorithm

The algorithm works as follows:

- When a non-leaf process in the spanning tree receives all the ack messages it expects, as well as a snap_completed message from each of its child processes, it sends a snap_completed message to its parent process.

Venkatesan's incremental snapshot algorithm

The algorithm works as follows:

- The algorithm terminates when the initiator has received all the ack messages it expects, as well as a snap_completed message from each of its child processes.
- The selective manner in which regular messages are sent has the effect that a process does not know whether to expect a regular message on an incoming channel.
- A process can be sure that no such message will be received and that the snapshot is complete only when it executes the "marker sending rule" for the next initiation of the algorithm

Snapshot algorithms for FIFO channels

1. Chandy-Lamport algorithm

Variations of the Chandy–Lamport algorithm

1. Spezialetti–Kearns algorithm
2. Venkatesan's incremental snapshot algorithm
3. Helary's wave synchronization method

Helary's wave synchronization method

Helary's snapshot algorithm incorporates the concept of message waves in the Chandy–Lamport algorithm.

A wave is a flow of control messages such that every process in the system is visited exactly once by a wave control message, and at least one process in the system can determine when this flow of control messages terminates.

A wave is initiated after the previous wave terminates.

Wave sequences may be implemented by various traversal structures such as a ring.

A process begins recording the local snapshot when it is visited by the wave control message.

Helary's wave synchronization method

In Helary's algorithm, the "marker sending rule" is executed when a control message belonging to the wave flow visits the process.

The process then forwards a control message to other processes, depending on the wave traversal structure, to continue the wave's progression.

Helary's wave synchronization method

The "marker receiving rule" is modified so that if the process has not recorded its state when a marker is received on some channel, the "marker receiving rule" is not executed and no messages received after the marker on this channel are processed until the control message belonging to the wave flow visits the process.

Thus, each process follows the "marker receiving rule" only after it is visited by a control message belonging to the wave.

Snapshot algorithms for FIFO channels

Conclusion

FIFO system ensures that all messages sent after a marker on a channel will be delivered after the marker.

This ensures that condition C2 is satisfied in the recorded snapshot if LSi, LSj, and SCij are recorded as described in the Chandy–Lamport algorithm.

UNIT 2

Global state and snapshot recording algorithms: (Chapter 4 -4.1 to 4.7)

- Introduction,
- System model and definitions,
- Snapshot algorithms for FIFO channels,
- Variations of the Chandy–Lamport algorithm,
- Snapshot algorithms for non-FIFO channels,
- Snapshots in a causal delivery system,
- Monitoring global state.

Snapshot algorithms for non-FIFO channels

- The problem of global snapshot recording is complicated because a marker cannot be used to delineate messages into those to be recorded in the global state from those not to be recorded in the global state.
- In such systems, different techniques have to be used to ensure that a recorded global state satisfies condition C2

Snapshot algorithms for non-FIFO channels

- In a non-FIFO system, to record a consistent global snapshot requirements are:

1. Degree of inhibition (i.e., temporarily delaying the execution of an application process or delaying the send of a computation message)
2. Piggybacking of control information on computation messages to capture out-of-sequence messages

List of Non-FIFO algorithms:

- Lai–Yang algorithm, Li *et al.* and Mattern – uses message piggybacking
- Helary - uses message inhibition

Lai–Yang algorithm

Lai and Yang's global snapshot algorithm for non-FIFO systems is based on two observations on the role of a marker in a FIFO system.

1. The first observation is that a marker ensures that condition C2 is satisfied for LSi and LSj when the snapshots are recorded at processes pi and pj, respectively.
2. The second observation is that the marker informs process pj of the value of $\{ send(mij) \mid send(mij) \in LSi \}$ so that the state of the channel Cij can be computed as transit(Lsi, LSj)

Lai–Yang algorithm

Lai and Yang's global snapshot algorithm for non-FIFO systems is based on two observations on the role of a marker in a FIFO system.

1. The first observation is that a marker ensures that condition C2 is satisfied for LSi and LSj when the snapshots are recorded at processes pi and pj, respectively.
2. The second observation is that the marker informs process pj of the value of $\{ send(mij) \mid send(mij) \in LSi \}$ so that the state of the channel Cij can be computed as transit(Lsi, LSj)

Lai–Yang algorithm

The first observation

The Lai-Yang algorithm fulfills this role of a marker in a non-FIFO system by using a coloring scheme on computation messages that works as follows:

1. Every process is initially white and turns red while taking a snapshot. The equivalent of the "Marker Sending Rule" is executed when a process turns red.
2. Every message sent by a white (red) process is colored white (red).
3. Thus, a white (red) message is a message that was sent before (after) the sender of that message recorded its local snapshot.
4. Every white process takes its snapshot at its convenience, but no later than the instant it receives a red message.

Lai–Yang algorithm

The Second observation

4. Every white process records a history of all white messages sent or received by it along each channel.
5. When a process turns red, it sends these histories along with its snapshot to the initiator process that collects the global snapshot.
6. The initiator process evaluates $transit(LS_i, LS_j)$ to compute the state of a channel C_{ij} as given below:
   SC_{ij} = white messages sent by p_i on C_{ij} − white messages received by p_j on C_{ij}
   $= \{send(m_{ij}) | send(m_{ij}) \in LS_i \} - \{rec(m_{ij}) | rec(m_{ij}) \in LS_j \}$.

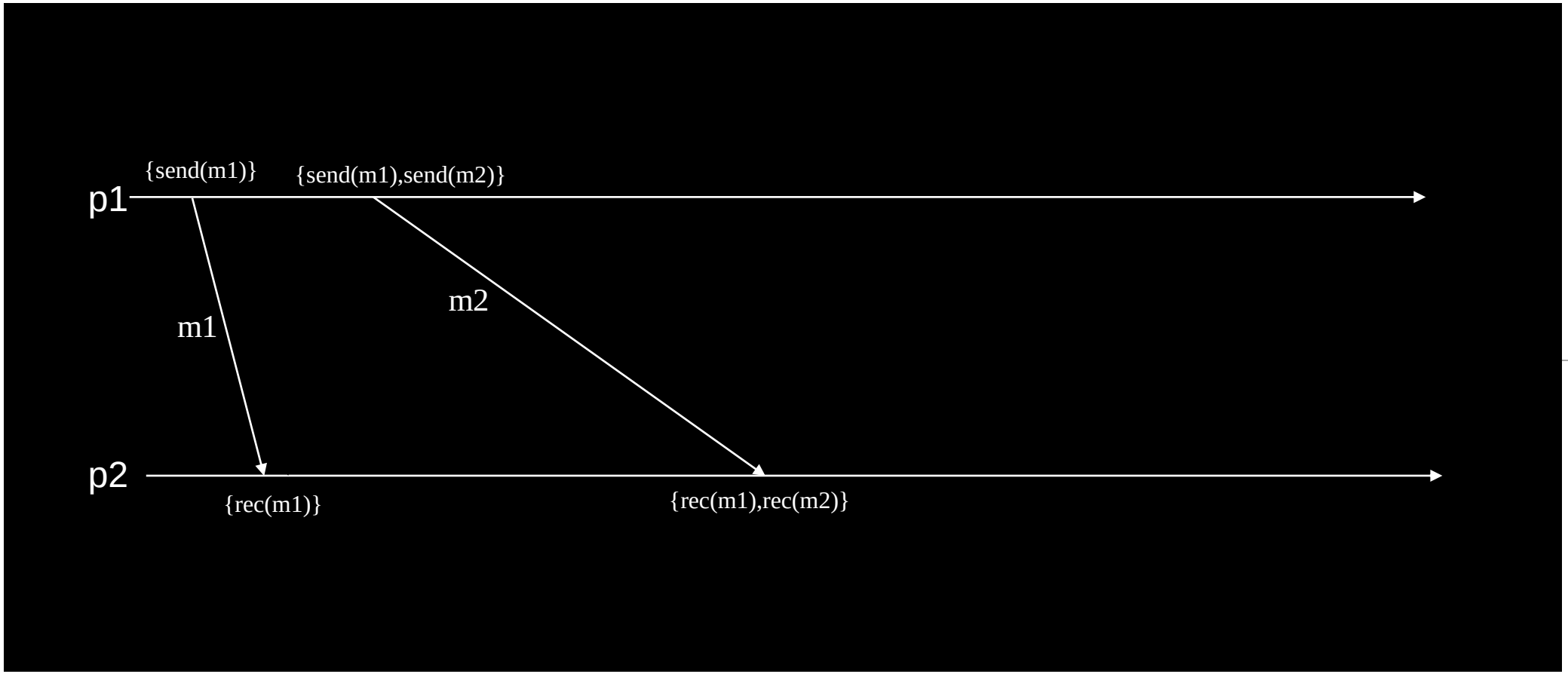
https://www.moovly.com/gallery/video/snapshot-lai-yang-algorithm/440cf11e-bee2-11eb-a69f-0699ff9171cb

Example

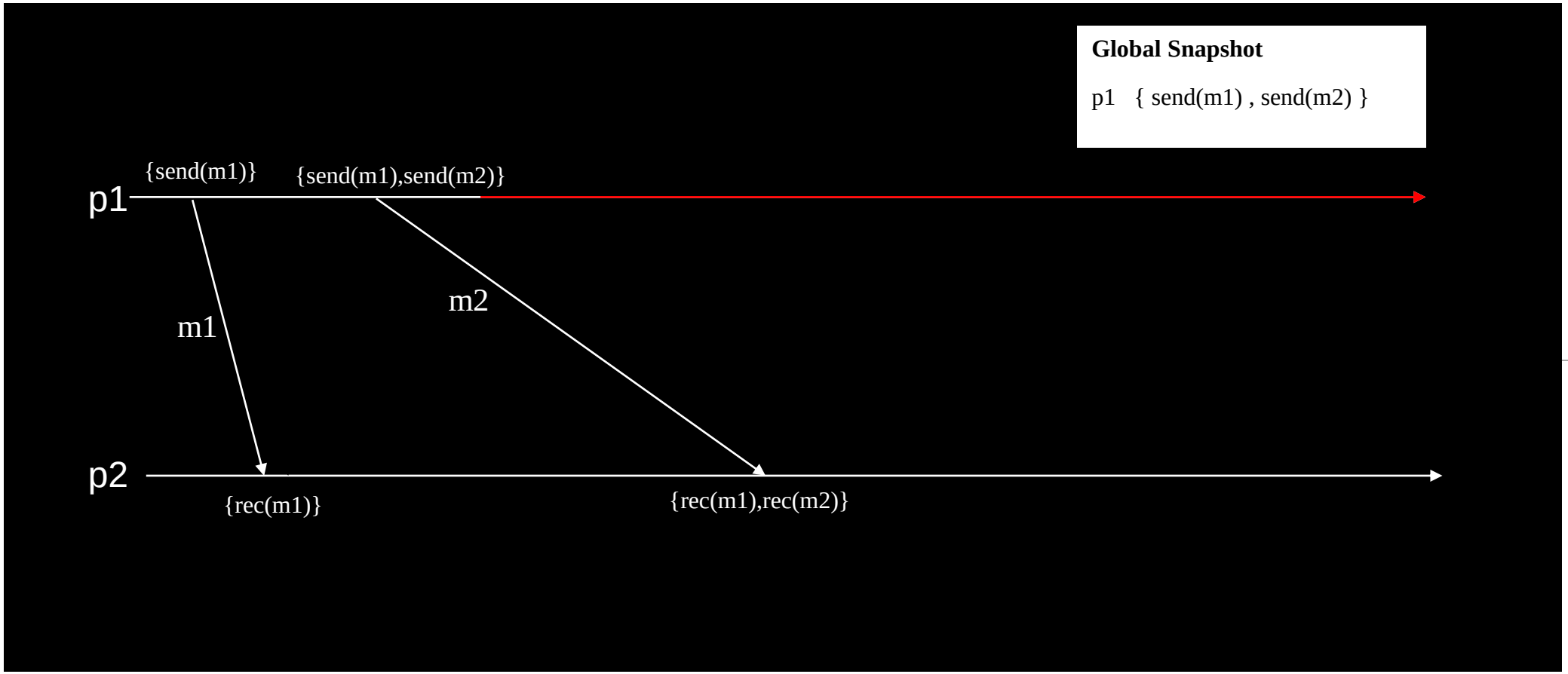
Consider 2 process p1 and p2 as shown in the timestamp diagram.

Process p1 and p2 are initially white

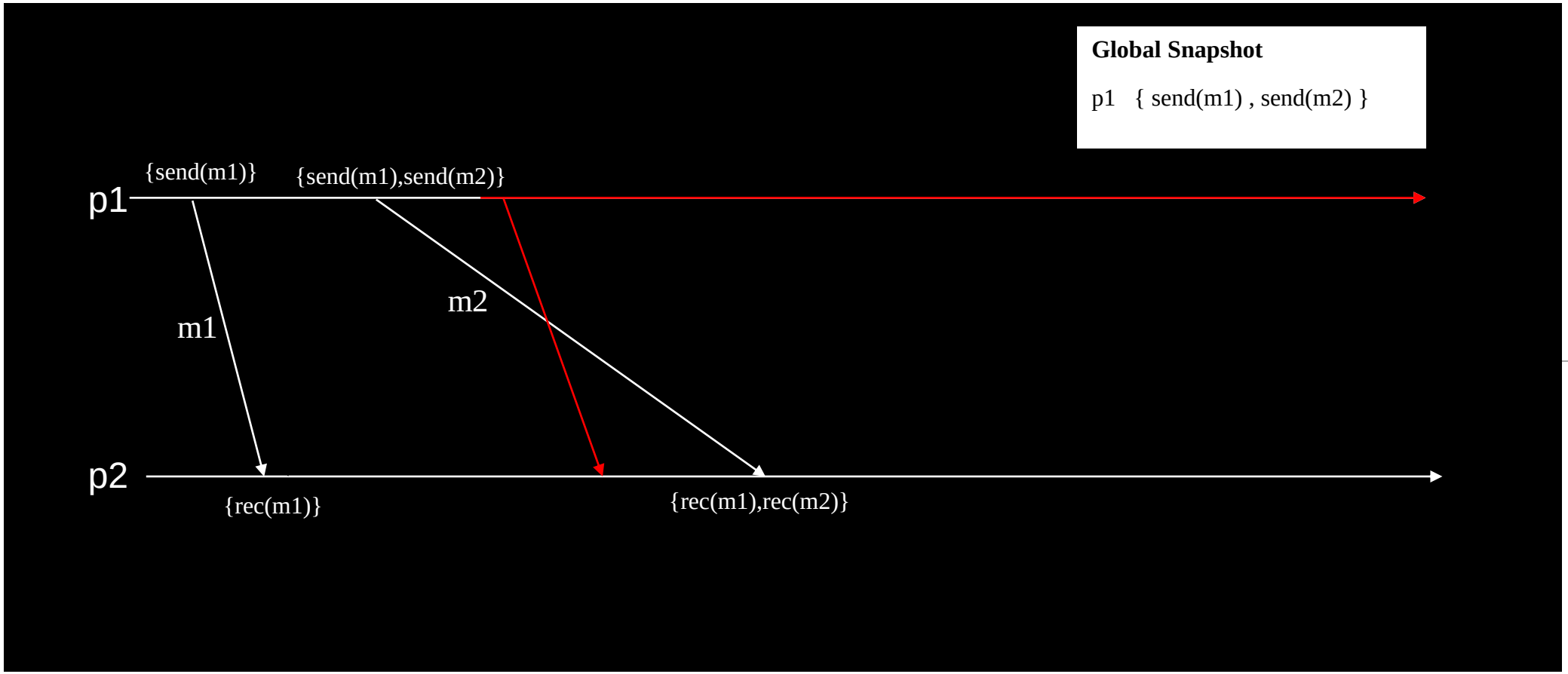
A message sent by a white process is colored white
Process p1 sends message m1,m2 to process p2.
Every white process records a history of all white messages sent or received by it along each channel.



Process p1 turns red while taking a snapshot.



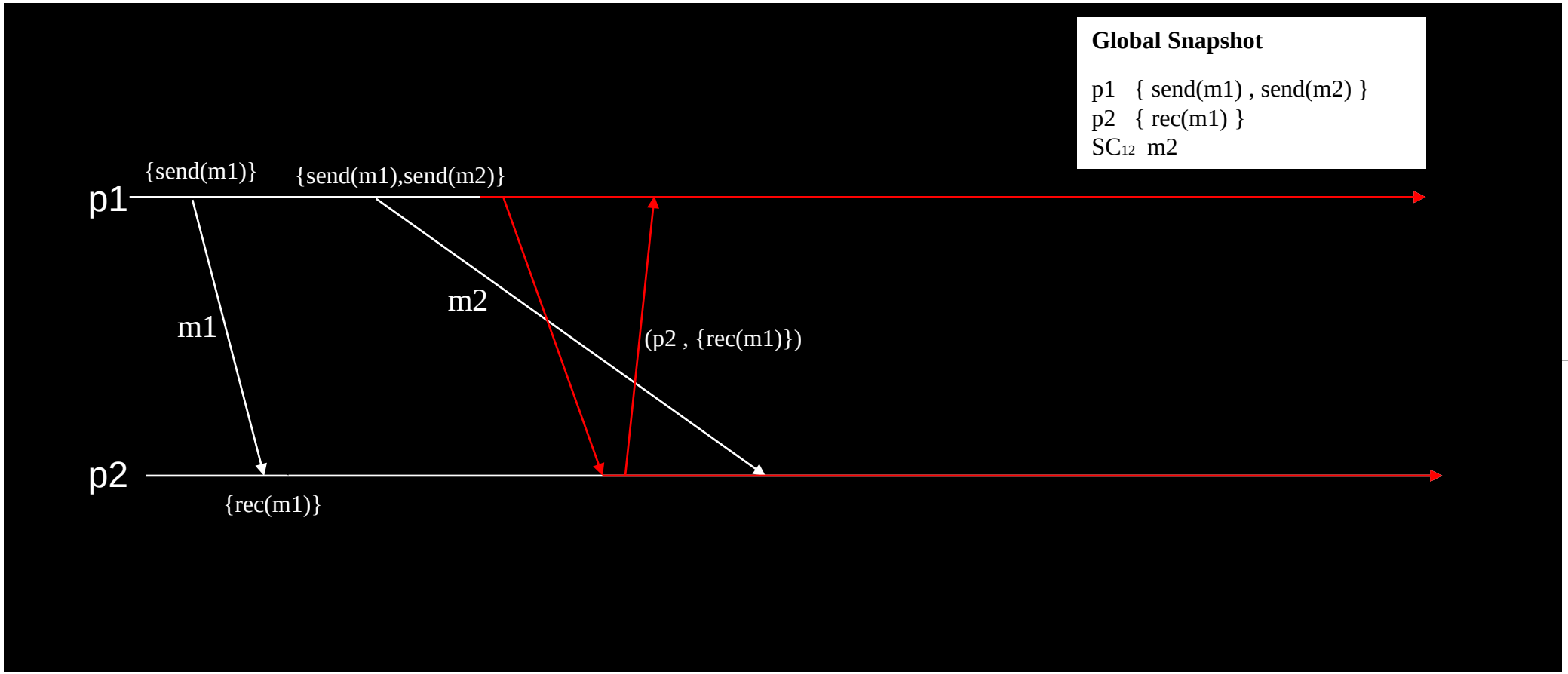
The equivalent of the “Marker Sending rule is executed when a process turns red



Process p2 takes its snapshot the instant it receives a red message and sends its snapshot and histories to initiator process.

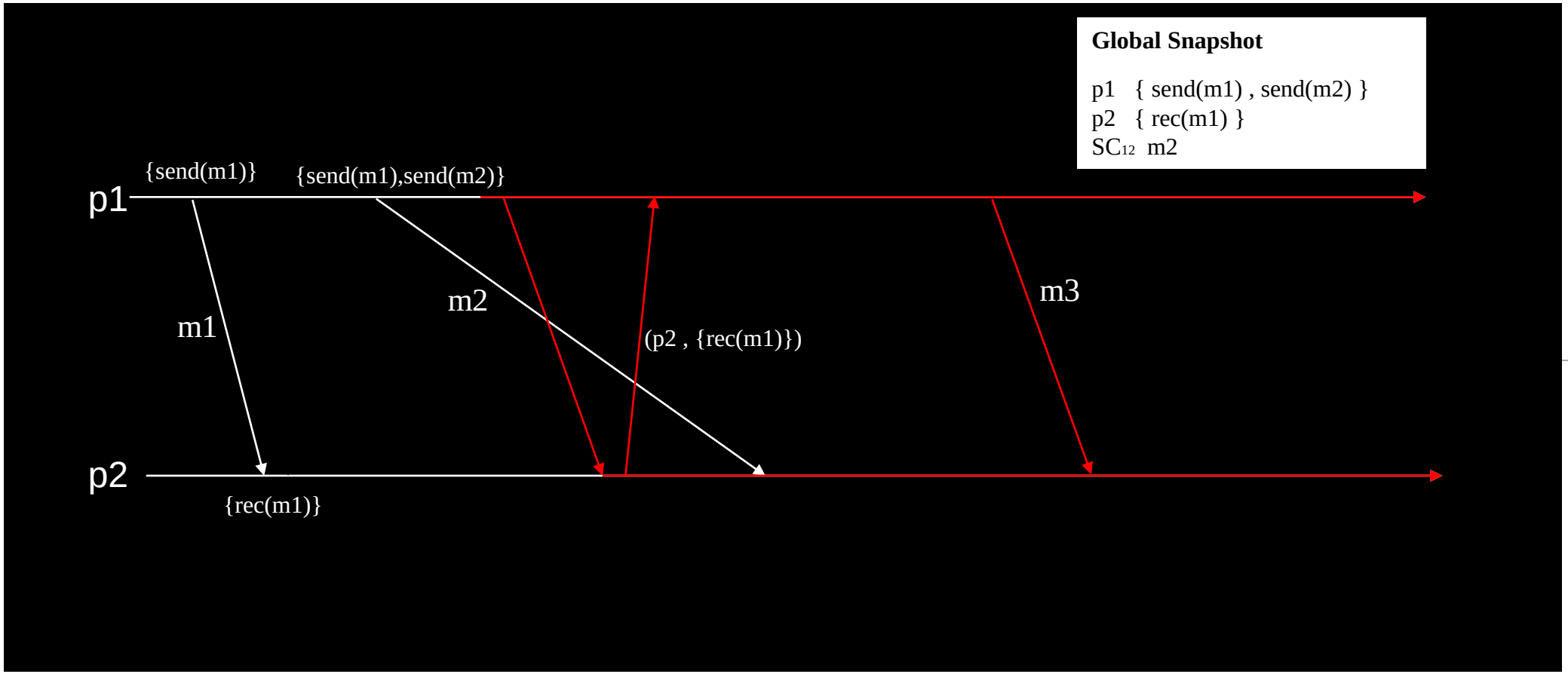
SCij = white messages sent by pi on Cij – white messages received by pj on Cij

= {send(mij)|send(mij ∈ LSi)} - {rec(mij)|rec(mij) ∈ LSj}



RAMAIAH
Institute of Technology

Process p1 send message m3 to process p2
A message sent by a red process is colored red.



Mattern's Algorithm for Non-FIFO channels

Mattern's algorithm is based on vector clocks and assumes a single initiator process and works as follows:

1. The initiator "ticks" its local clock and selects a future vector time s at which it would like a global snapshot to be recorded. It then broadcasts this time s and freezes all activity until it receives all acknowledgements of the receipt of this broadcast.
2. When a process receives the broadcast, it remembers the value s and returns an acknowledgement to the initiator.
3. After having received an acknowledgement from every process, the initiator increases its vector clock to s and broadcasts a dummy message to all processes.
4. The receipt of this dummy message forces each recipient to increase its clock to a value $\geq s$ if not already $\geq s$.
5. Each process takes a local snapshot and sends it to the initiator when (just before) its clock increases from a value less than s to a value $\geq s$.
6. The state of C_{ij} is all messages sent along C_{ij}, whose timestamp is smaller than s and which are received by p_j after recording LS_j.

Mattern's Algorithm

- A termination detection scheme for non-FIFO channels is required to detect that no white messages are in transit.
- One of the following schemes can be used for termination detection:

First method:

- Each process i keeps a counter $cntr_i$ that indicates the difference between the number of white messages it has sent and received before recording its snapshot.
- It reports this value to the initiator process along with its snapshot and forwards all white messages, it receives henceforth, to the initiator.
- Snapshot collection terminates when the initiator has received $\sum_i cntr_i$ number of forwarded white messages.

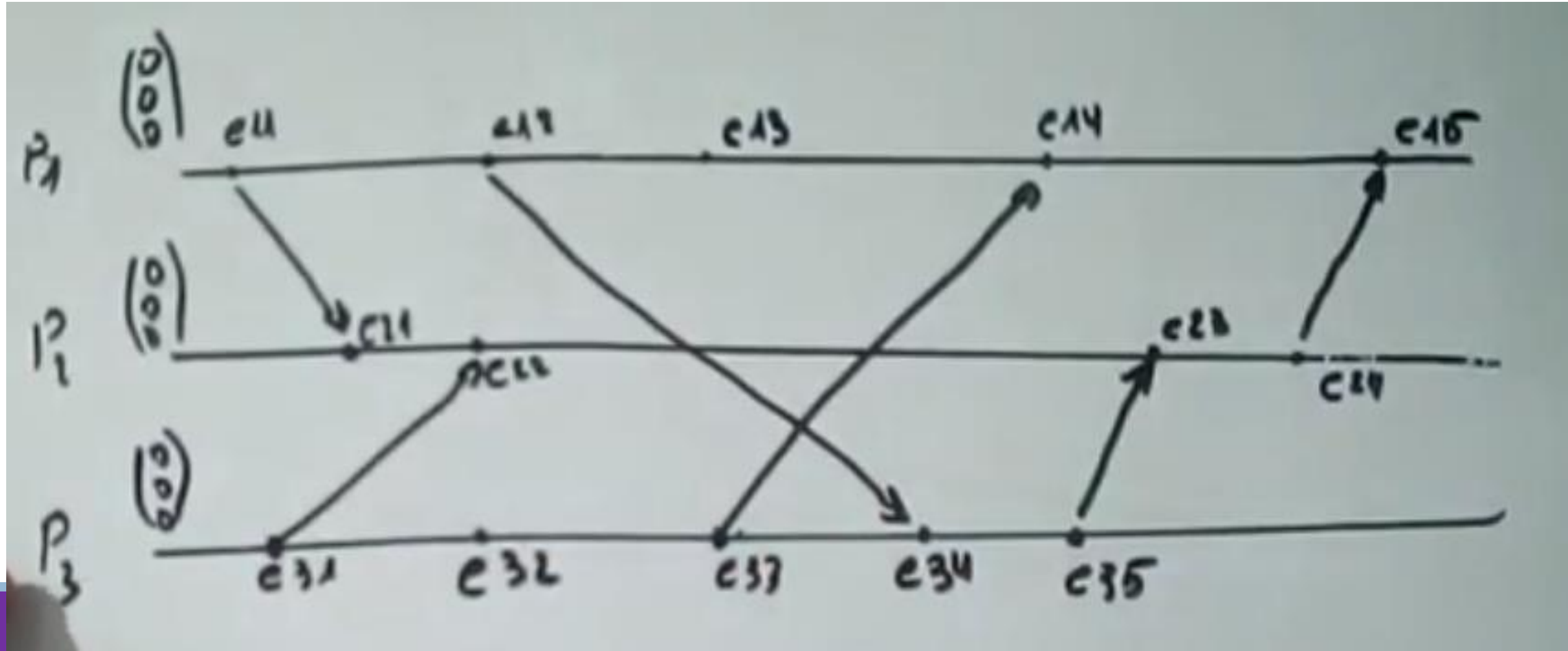
Mattern's Algorithm

Second method:

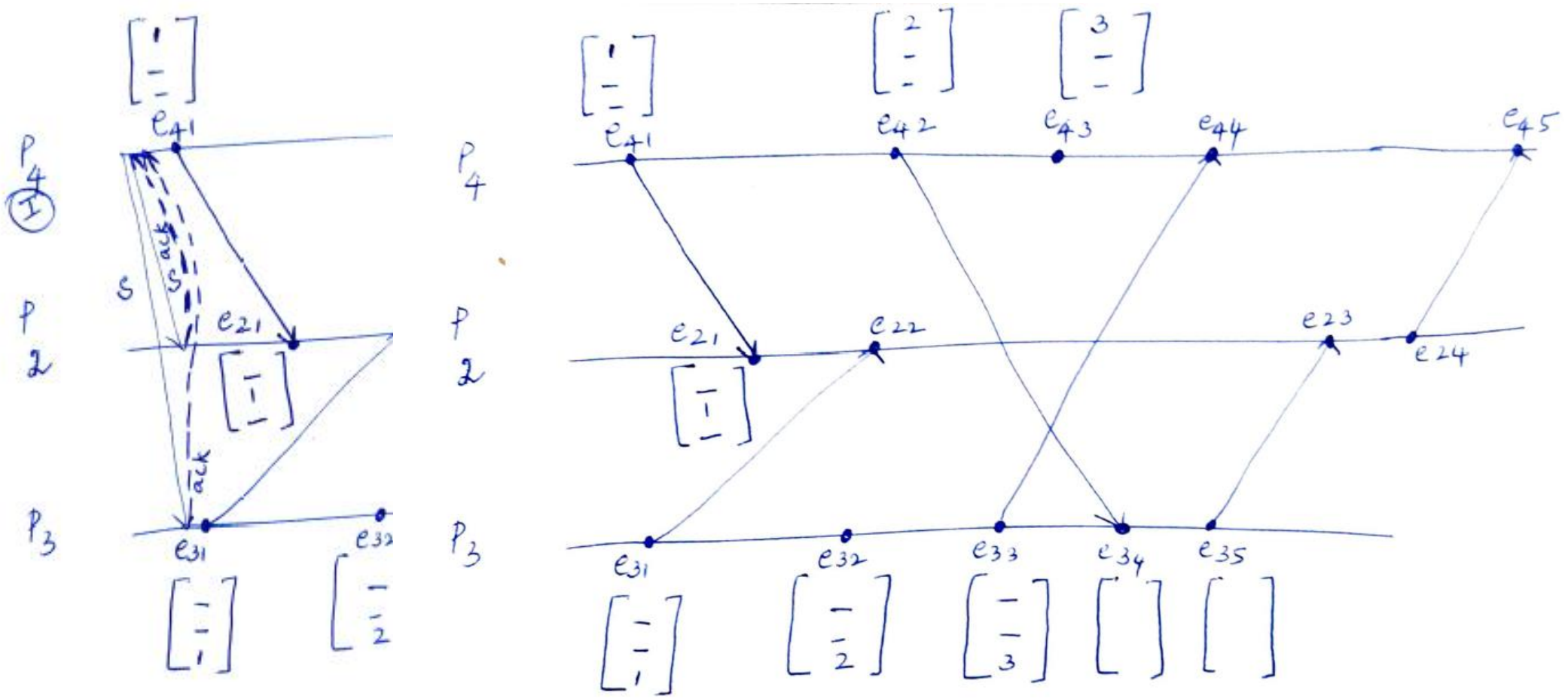
- Each red message sent by a process carries a piggybacked value of the number of white messages sent on that channel before the local state recording.
- Each process keeps a counter for the number of white messages received on each channel.
- A process can detect termination of recording the states of incoming channels when it receives as many white messages on each channel as the value piggybacked on red messages received on that channel.

RAMAIAH
Institute of Technology

Mattern's Algorithm example

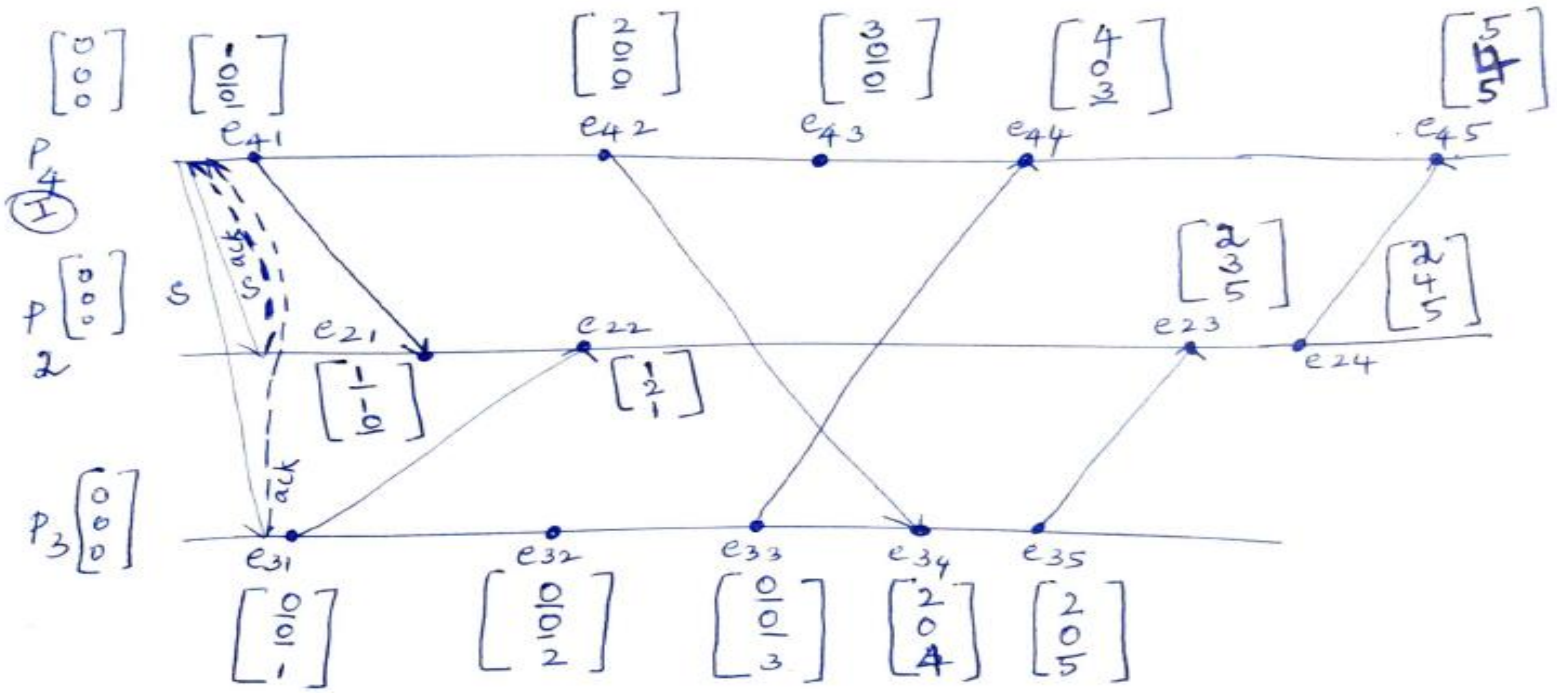


Mattern's Algorithm example



RAMAIAH Institute of Technology

Mattern's Algorithm example



$e_2 \rightarrow e_1$ $\begin{pmatrix} e_1 \\ t_1 \\ t_2 \\ t_3 \end{pmatrix} \geq \begin{pmatrix} e_2 \\ t_1 \\ t_2 \\ t_3 \end{pmatrix}$

$\begin{pmatrix} e_1 \\ t_1 \\ t_2 \\ t_3 \end{pmatrix} \; // \; \begin{pmatrix} e_2 \\ t_1 \\ t_2 \\ t_3 \end{pmatrix}$ ($>$, $<$, $=/>)

parallel

$\begin{pmatrix} e_1 \\ t_1 \\ t_2 \\ t_3 \end{pmatrix} \; // \; \begin{pmatrix} e_2 \\ t_1 \\ t_2 \\ t_3 \end{pmatrix}$ ($=$)

Snapshots in a causal delivery system

- The causal message delivery property **CO** provides a built-in message synchronization to control and computation messages.
- Two global snapshot recording algorithms, namely, Acharya-Badrinath and Alagar-Venkatesan exist that assume that the underlying system supports causal message delivery.

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

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

Snapshots in a causal delivery system

- In both these algorithms recording of process state is identical and proceed as follows :
- An initiator process broadcasts a token, denoted as *token*, to every process including itself.
- Let the copy of the token received by process p_i be denoted $token_i$.
- A process p_i records its local snapshot LS_i when it receives $token_i$ and sends the recorded snapshot to the initiator.
- The algorithm terminates when the initiator receives the snapshot recorded by each process.
- Channel state recording is different in these two algorithms

Snapshots in a causal delivery system

Correctness

For any two processes p_i and p_j, the following property is satisfied:

$$send(m_{ij}) \notin LS_i \Rightarrow rec(m_{ij}) \notin LS_j.$$

- This is due to the causal ordering property of the underlying system as explained next.
  - Let a message m_{ij} be such that $rec(token_i) \longrightarrow send(m_{ij})$.
  - Then $send(token_j) \longrightarrow send(m_{ij})$ and the underlying causal ordering property ensures that $rec(token_j)$, at which instant process p_j records LS_j, happens before $rec(m_{ij})$.
  - Thus, m_{ij} whose send is not recorded in LS_i, is not recorded as received in LS_j.

Snapshots in a causal delivery system

Complexity:

- This algorithm requires $2n$ messages and 2 time units for recording and assembling the snapshot, where one time unit is required for the delivery of a message.
- If the contents of messages in channels state are required, the algorithm requires $2n$ messages and 2 time units additionally.

Channel state recording in Acharya–Badrinath algorithm

- Each process p_i maintains arrays $SENT_i[1, ...N]$ and $RECD_i[1, ..., N]$.
- $SENT_i[j]$ is the number of messages sent by process p_i to process p_j.
- $RECD_i[j]$ is the number of messages received by process p_i from process p_j.

Snapshots in a causal delivery system

- Channel states are recorded as follows:
  When a process p_i records its local snapshot LS_i on the receipt of $token_i$, it includes arrays $RECD_i$ and $SENT_i$ in its local state before sending the snapshot to the initiator.

When the algorithm terminates, the initiator determines the state of channels as follows:

- The state of each channel from the initiator to each process is empty.
- The state of channel from process p_i to process p_j is the set of messages whose sequence numbers are given by $\{RECD_j[i] + 1, \ldots, SENT_i[j]\}$.

Channel state recording in Alagar-Venkatesan algorithm

- A message is referred to as *old* if the send of the message causally precedes the send of the token.
- Otherwise, the message is referred to as *new*.

In Alagar-Venkatesan algorithm channel states are recorded as follows:

1. When a process receives the *token*, it takes its snapshot, initializes the state of all channels to empty, and returns *Done* message to the initiator. Now onwards, a process includes a message received on a channel in the channel state only if it is an old message.
2. After the initiator has received *Done* message from all processes, it broadcasts a *Terminate* message.
3. A process stops the snapshot algorithm after receiving a *Terminate* message.

Algorithms	Features
Chandy-Lamport	Baseline algorithm. Requires FIFO channels. $O(e)$ messages to record snapshot and $O(d)$ time.
Spezialetti-Kearns	supports concurrent initiators, efficient assembly and distribution of a snapshot. Assumes bidirectional channels. $O(e)$ messages to record, $O(rn^2)$ messages to assemble and distribute snapshot.
Lai-Yang	Works for non-FIFO channels. Markers piggybacked on computation messages. Message history required to compute channel states.
Li et al.	Small message history needed as channel states are computed incrementally.
Mattern	No message history required. Termination detection required to compute channel states.
Acharya-Badrinath	Requires causal delivery support, Centralized computation of channel states, Channel message contents need not be known. Requires 2n messages, 2 time units.
Alagar-Venkatesan	Requires causal delivery support, Distributed computation of channel states. Requires $3n$ messages, 3 time units, small messages.

n = # processes, u = # edges on which messages were sent after previous snapshot, e = # channels, d is the diameter of the network, r = # concurrent initiators.

Monitoring global state

- Several applications such as debugging a distributed program need to detect a system state which is determined by the values of variables on a subset of processes.
- Rather than recording and evaluating snapshots at regular intervals, it is more efficient to monitor changes to the variables
- The monitor evaluates the global predicate when it receives the next message from each of the involved processes, informing it of the value(s) of their local variable(s).

RAMAIAH Institute of Technology

UNIT 2

Terminology and basic algorithms:

(Chapter 5-5.1,5.2, 5.3,5.4,5.5,5.6,5.7,5.10)

- Topology abstraction and overlays
- Classifications and basic concepts
- Complexity measures and metrics
- Program structure
- Elementary graph algorithms
- Synchronizers
- Maximal independent set (MIS)
- Leader election

RAMAIAH
Institute of Technology

Topology abstraction and overlays

- The topology of a distributed system can be typically **viewed as an undirected graph** in which the **nodes represent the processors** and the **edges represent the links connecting the processors**.
- Physical - arrangement of devices on a computer network through the actual cables that transmit data.
- Logical- routers and switches, communication mechanism for all nodes in a network.

RAMAIAH
Institute of Technology

Topology abstraction and overlays

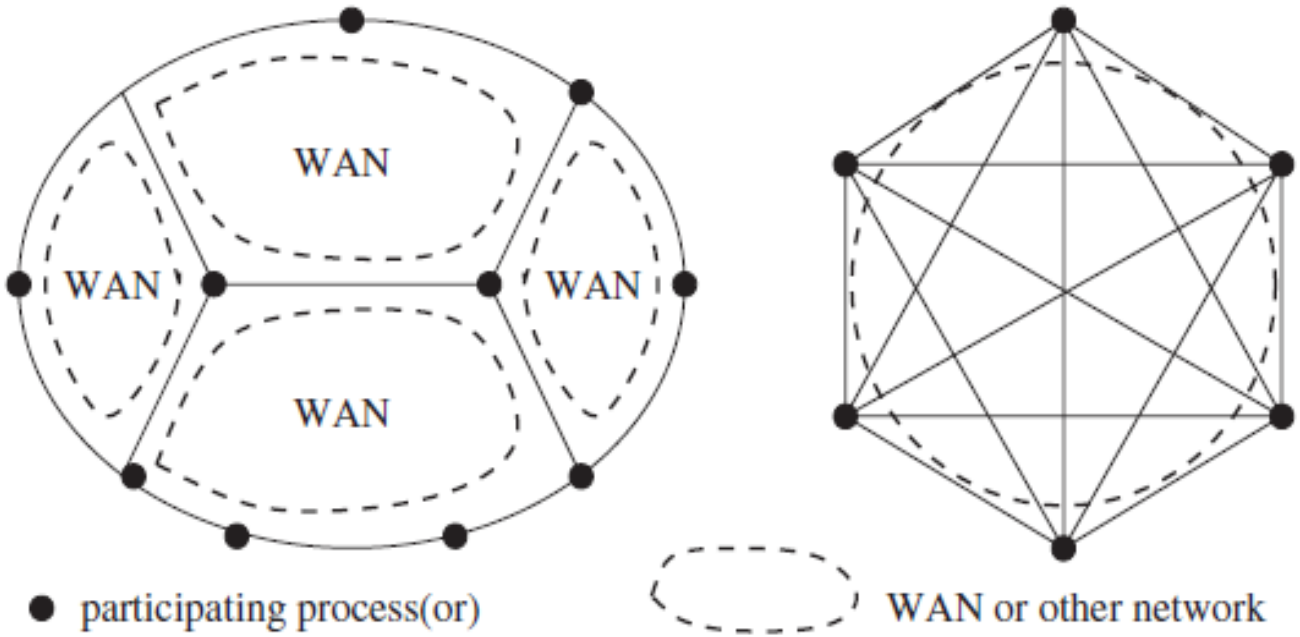


Figure 5.1 Two examples of topological views at different levels of abstraction.

In Figure 5.1(a), the physical topology is not shown explicitly to keep the figure simple.

Figure 5.1(b) shows each pair of nodes in the logical topology is connected to give a fully connected network.

RAMAIAH
Institute of Technology

Topology abstraction and overlays

- Superimposed- peer-to-peer networks and client-server applications are overlay networks because their nodes run on top of the Internet

Notation

Whatever the level of topological view we are dealing with, we assume that an undirected graph (N, L) is used to represent the topology. The notation $n = |N|$ and $l = |L|$ will also be used.

Classifications and basic concepts

- Application executions and control algorithm executions
- Centralized and distributed algorithms
- Symmetric and asymmetric algorithms
- Anonymous algorithms
- Uniform algorithms
- Adaptive algorithms
- Execution inhibition
- Synchronous and asynchronous systems
- Online versus offline algorithms
- Failure models
- Wait-free algorithms
- Communication channels

RAMAIAH Institute of Technology

Classifications and basic concepts

Application executions and control algorithm executions

- The distributed application execution is comprised of the execution of instructions, including the communication instructions, within the distributed application program.
- The application execution represents the logic of the application.
- Control algorithm also needs to be executed in order to monitor the application execution or to perform various auxiliary functions.
- The control algorithm performs functions such as: creating a spanning tree, creating a connected dominating set, achieving consensus among the nodes, distributed transaction commit, distributed deadlock detection, global predicate detection, termination detection, global state recording, checkpointing, and also memory consistency enforcement in distributed shared memory systems.

RAMAIAH Institute of Technology

Classifications and basic concepts

Centralized and distributed algorithms

Centralized algorithm is one in which a predominant amount of work is performed by one (or possibly a few) processors, whereas other processors play a relatively smaller role in accomplishing the joint task.

Example : client–server

A distributed algorithm is one in which each processor plays an equal role in sharing the message overhead, time overhead, and space overhead. It is difficult to design a purely distributed algorithm (that is also efficient) for some applications.

Consider the problem of recording a global state of all the nodes.

Classifications and basic concepts

Symmetric and asymmetric algorithms

A symmetric algorithm is an algorithm in which all the processors execute the same logical functions.

An asymmetric algorithm is an algorithm in which different processors execute logically different (but perhaps partly overlapping) functions.

- A centralized algorithm is always asymmetric.
- An algorithm that is not fully distributed is also asymmetric.
- In the client–server configuration, the clients and the server execute asymmetric algorithms

Classifications and basic concepts

Anonymous algorithms

An anonymous system is a system in which neither processes nor processors use their process identifiers and processor identifiers to make any execution decisions in the distributed algorithm.

An anonymous algorithm is an algorithm which runs on an anonymous system and therefore does not use process identifiers or processor identifiers in the code.

An anonymous algorithm possesses structural elegance.

Classifications and basic concepts

Uniform algorithms

A uniform algorithm is an algorithm that does not use n, the number of processes in the system, as a parameter in its code.

A uniform algorithm is desirable because it allows scalability transparency, and processes can join or leave the distributed execution without intruding on the other processes, except its immediate neighbors that need to be aware of any changes in their immediate topology.

Algorithms that run on a logical ring and have nodes communicate only with their neighbors are uniform.

Classifications and basic concepts

Adaptive algorithms

An adaptive algorithm is **an algorithm that changes its behavior at the time it is run, based on information available and on a priori defined reward mechanism (or criterion)**.

Example : Routing Algorithm

Classifications and basic concepts

Deterministic versus non-deterministic executions

A deterministic receive primitive specifies the source from which it wants to receive a message.

A non-deterministic receive primitive can receive a message from any source – the message delivered to the process is the first message that is queued in the local incoming buffer, or the first message that comes in subsequently if no message is queued in the local incoming buffer.

A distributed program that contains no non-deterministic receives has a deterministic execution; otherwise, if it contains at least one non-deterministic receive primitive, it is said to have a non-deterministic execution.

Classifications and basic concepts

Execution inhibition

Inhibition refers to protocols delaying actions of the underlying system execution for an interval of time.

Protocols that require processors to suspend their normal execution until some series of actions stipulated by the protocol have been performed are termed as inhibitory or freezing protocols

An orthogonal classification is that of *send inhibition*, *receive inhibition*, and *internal event inhibition*:

- A protocol is *send inhibitory* if some delayed events are send events.
- A protocol is *receive inhibitory* if some delayed events are receive events.
- A protocol is *internal event inhibitory* if some delayed events are internal events.

Classifications and basic concepts

Synchronous and asynchronous systems

A synchronous system is a system that satisfies the following properties:

- There is a known upper bound on the message communication delay.
- There is a known bounded drift rate for the local clock of each processor with respect to real-time. The drift rate between two clocks is defined as the rate at which their values diverge.
- There is a known upper bound on the time taken by a process to execute a logical step in the execution.

Classifications and basic concepts

Synchronous and asynchronous systems

An asynchronous system is a system in which none of the above three properties of synchronous systems are satisfied.

Clearly, systems can be designed that satisfy some combination but not all of the criteria that define a synchronous system.

Classifications and basic concepts

Online versus offline algorithms

An on-line algorithm is an algorithm that executes as the data is being generated.

An off-line algorithm is an algorithm that requires all the data to be available before algorithm execution begins.

Clearly, on-line algorithms are more desirable.

Classifications and basic concepts

Failure models

A failure model specifies the manner in which the component(s) of the system may fail.

There exists a rich class of well-studied failure models

1. **Fail-stop**
2. **Crash**
3. **Receive omission**
4. **Send omission**
5. **General omission**
6. **Byzantine or malicious failure, with authentication**
7. **Byzantine or malicious failure**

Classifications and basic concepts

Wait-free algorithms

A *wait-free algorithm* is an algorithm that can execute (synchronization operations) in an (n−1) process fault tolerant manner, i.e., it is resilient to n−1 process failures.

Thus, if an algorithm is wait-free, then the (synchronization) operations of any process must complete in a bounded number of steps irrespective of the failures of all the other processes.

Classifications and basic concepts

Communication channels

Communication channels are normally first-in first-out queues (FIFO).

At the network layer, this property may not be satisfied, giving non-FIFO channels.

Complexity measures and metrics

Space complexity per node This is the memory requirement at a node. The best case, average case, and worst case memory requirement at a node can be specified.

System wide space complexity The system space complexity (best case, average case, or worst case) is not necessarily n times the corresponding space complexity (best case, average case, or worst case) per node.

Time complexity per node This measures the processing time per node, and does not explicitly account for the message propagation/transmission times, which are measured as a separate metric.

System wide time complexity If the processing in the distributed system occurs at all the processors concurrently, then the system time complexity is not n times the time complexity per node.

Program structure

- Hoare, who pioneered programming language support for concurrent processes, designed concurrent sequential processes (CSP), which allows communicating processes to synchronize efficiently.
- The typical program structure for any process in a distributed application is based on CSP's repetitive command over the alternative command on multiple guarded commands, and is as follows:

Program structure

The repetitive command (denoted by "*") denotes an infinite loop.

Inside the repetitive command is the alternative command over guarded commands.

The alternative command, denoted by a sequence of "| |" separating guarded commands, specifies execution of exactly one of its constituent guarded commands.

$$*[\,G_1 \longrightarrow CL_1 \,||\, G_2 \longrightarrow CL_2 \,||\, \cdots \,||\, G_k \longrightarrow CL_k\,].$$

RAMAIAH Institute of Technology

Program structure

The guarded command has the syntax "G −→ CL"

where

- guard G is a boolean expression

The guard expression may contain a term to check if a message from any other process has arrived.

- CL is a list of commands that are only executed if G is true.

$$*[\, G_1 \longrightarrow CL_1 \,||\, G_2 \longrightarrow CL_2 \,|| \cdots || \, G_k \longrightarrow CL_k \,].$$

```
(local variables)
int parent ←— ⊥
set of int Children, Unrelated ←— ∅
set of int Neighbors ←— set of neighbors
(message types)
QUERY, ACCEPT, REJECT

(1)   When the predesignated root node wants to initiate the algorithm:
(1a)  if (i = root and parent = ⊥) then
(1b)      send QUERY to all neighbors;
(1c)      parent ←— i.

(2)   When QUERY arrives from j:
(2a)  if parent = ⊥ then
(2b)      parent ←— j;
(2c)      send ACCEPT to j;
(2d)      send QUERY to all neighbors except j;
(2e)      if (Children ∪ Unrelated) = (Neighbors/{parent}) then
(2f)          terminate.
(2g)  else send REJECT to j.

(3)   When ACCEPT arrives from j:
(3a)  Children ←— Children ∪ {j};
(3b)  if (Children ∪ Unrelated) = (Neighbors/{parent}) then
(3c)      terminate.

(4)   When REJECT arrives from j:
(4a)  Unrelated ←— Unrelated ∪ {j};
(4b)  if (Children ∪ Unrelated) = (Neighbors/{parent}) then
(4c)      terminate.
```

Algorithm 5.2 Spanning tree algorithm: the asynchronous algorithm assuming a designated root that initiates a flooding. The code shown is for processor P_i, $1 \leq i \leq n$.

RAMAIAH
Institute of Technology

Program structure

The format for the pseudo-code used

1. The process-local variables whose scope is global to the process, and message types, are declared first.
2. Shared variables, if any, (for distributed shared memory systems) are explicitly labeled as such.
3. This is followed by any initialization code.
4. The *repetitive* and the *alternative* commands are not explicitly shown.
5. The *guarded* commands are shown as explicit modules or procedures (e.g., lines 1–4 in Algorithm 5.2). The guard usually checks for the arrival of a message of a certain type, perhaps with additional conditions on some parameter values and other local variables.
6. The body of the procedure gives the list of commands to be executed if the guard evaluates to *true*.
7. Process termination may be explicitly stated in the body of any procedure(s).
8. The symbol $\perp$ is used to denote an undefined value. When used in a comparison, its value is $-\infty$.

```
(local variables)
int parent ← ⊥
set of int Children, Unrelated ← ∅
set of int Neighbors ← set of neighbors
(message types)
QUERY, ACCEPT, REJECT

(1)   When the predesignated root node wants to initiate the algorithm:
(1a)  if (i = root and parent = ⊥) then
(1b)      send QUERY to all neighbors;
(1c)      parent ← i.

(2)   When QUERY arrives from j:
(2a)  if parent = ⊥ then
(2b)      parent ← j;
(2c)      send ACCEPT to j;
(2d)      send QUERY to all neighbors except j;
(2e)      if (Children ∪ Unrelated) = (Neighbors/{parent}) then
(2f)          terminate.
(2g)  else send REJECT to j.

(3)   When ACCEPT arrives from j:
(3a)  Children ← Children ∪ {j};
(3b)  if (Children ∪ Unrelated) = (Neighbors/{parent}) then
(3c)      terminate.

(4)   When REJECT arrives from j:
(4a)  Unrelated ← Unrelated ∪ {j};
(4b)  if (Children ∪ Unrelated) = (Neighbors/{parent}) then
(4c)      terminate.
```

Algorithm 5.2 Spanning tree algorithm: the asynchronous algorithm assuming a designated root that initiates a flooding. The code shown is for processor P_i, $1 \leq i \leq n$.

RAMAIAH
Institute of Technology

Elementary graph algorithms

Familiar with the centralized algorithms to solve basic graph problems.

The distributed algorithms find difficulty in designing distributed algorithms wherein each node has only a partial view of the graph (system), which is confined to its immediate neighbors.

- Node can communicate with only its immediate neighbors along the incident edges.
- Assume unweighted, undirected edges, and asynchronous execution by the processors.
- Communication is by message-passing on the edges.

RAMAIAH
Institute of Technology

Elementary graph algorithms

Elementary algorithms are theoretically important from a practical perspective because spanning trees are a very efficient form of information distribution and collection in distributed systems.

- The first algorithm is a synchronous spanning tree algorithm.
- The next three are asynchronous algorithms to construct spanning trees.

RAMAIAH
Institute of Technology

Elementary graph algorithms

- Synchronous spanning tree algorithm.
  - **Synchronous single-initiator spanning tree algorithm using flooding**
- Asynchronous algorithms to construct spanning trees.
  - **Asynchronous single-initiator spanning tree algorithm using flooding**

Synchronous single-initiator spanning tree algorithm using flooding

The code for all processes is not only symmetrical, but also proceeds in rounds.

```
(local variables)
int visited, depth ← 0
int parent ← ⊥
set of int Neighbors ← set of neighbors
(message types)
QUERY

(1)  if i = root then
(2)      visited ← 1;
(3)      depth ← 0;
(4)      send QUERY to Neighbors;
(5)  for round = 1 to diameter do
(6)      if visited = 0 then
(7)          if any QUERY messages arrive then
(8)              parent ← randomly select a node from which
                    QUERY was received;
(9)              visited ← 1;
(10)             depth ← round;
(11)             send QUERY to Neighbors \ {senders of
                    QUERYs received in this round};
(12)     delete any QUERY messages that arrived in this round.
```

Algorithm 5.1 Spanning tree algorithm: the synchronous breadth-first search (BFS) spanning tree algorithm. The code shown is for processor P_i, $1 \leq i \leq n$.

Synchronous single-initiator spanning tree algorithm using flooding

This algorithm assumes a designated root node, root, which initiates the algorithm.

The pseudo-code for each process Pi is shown in Algorithm 5.1.

The root initiates a flooding of QUERY messages in the graph to identify tree edges.

The parent of a node is that node from which a QUERY is first received; if multiple QUERYs are received in the same round, one of the senders is randomly chosen as the parent.

```
(local variables)
int visited, depth ← 0
int parent ← ⊥
set of int Neighbors ← set of neighbors
(message types)
QUERY

(1)  if i = root then
(2)      visited ← 1;
(3)      depth ← 0;
(4)      send QUERY to Neighbors;
(5)  for round = 1 to diameter do
(6)      if visited = 0 then
(7)          if any QUERY messages arrive then
(8)              parent ← randomly select a node from which
                    QUERY was received;
(9)              visited ← 1;
(10)             depth ← round;
(11)             send QUERY to Neighbors \ {senders of
                    QUERYs received in this round};
(12)     delete any QUERY messages that arrived in this round.
```

Algorithm 5.1 Spanning tree algorithm: the synchronous breadth-first search (BFS) spanning tree algorithm. The code shown is for processor P_i, $1 \leq i \leq n$.

(local variables)
int *visited*, *depth* $\longleftarrow 0$
int *parent* $\longleftarrow \perp$
set of int *Neighbors* $\longleftarrow$ set of neighbors
(message types)
QUERY

```
(1)   if i = root then
(2)         visited ⟵ 1;
(3)         depth ⟵ 0;
(4)         send QUERY to Neighbors;
(5)   for round = 1 to diameter do
(6)         if visited = 0 then
(7)               if any QUERY messages arrive then
(8)                     parent ⟵ randomly select a node from which
                          QUERY was received;
(9)                     visited ⟵ 1;
(10)                    depth ⟵ round;
(11)                    send QUERY to Neighbors \ {senders of
                          QUERYs received in this round};
(12)        delete any QUERY messages that arrived in this round.
```

Synchronous single-initiator spanning tree algorithm using flooding

- Figure 5.2 shows an example execution of the algorithm with node A as initiator.
- The resulting tree is shown in boldface,
- Round numbers in which the QUERY messages are sent are indicated next to the messages.
- For example, at the end of round 2, E receives a QUERY from B and F and randomly chooses F as the parent.
- A total of nine QUERY messages are sent in the network which has eight links.

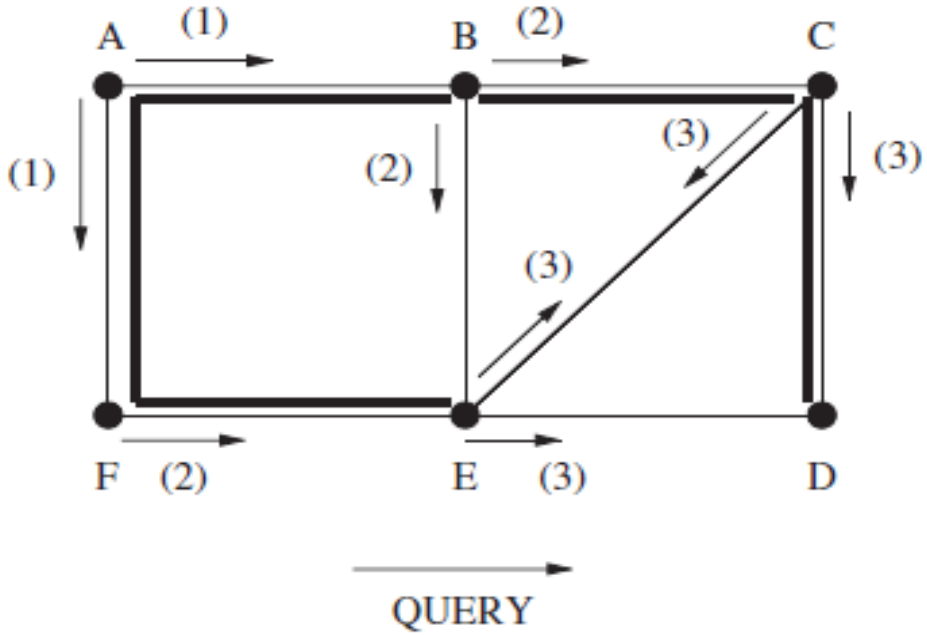


Figure 5.2 Example execution of the synchronous BFS spanning tree algorithm (Algorithm 5.1).

RAMAIAH
Institute of Technology

Asynchronous single-initiator spanning tree algorithm using flooding

- The pseudo-code for each process Pi is shown in Algorithm 5.2.
- In asynchronous system, there is no bound on the time it takes to propagate a message, and hence no notion of a message round.

```
(1)   When the predesignated root node wants to initiate the algorithm:
(1a)  if (i = root and parent = ⊥) then
(1b)      send QUERY to all neighbors;
(1c)      parent ← i.

(2)   When QUERY arrives from j:
(2a)  if parent = ⊥ then
(2b)      parent ← j;
(2c)      send ACCEPT to j;
(2d)      send QUERY to all neighbors except j;
(2e)      if (Children ∪ Unrelated) = (Neighbors/{parent}) then
(2f)          terminate.
(2g)  else send REJECT to j.

(3)   When ACCEPT arrives from j:
(3a)  Children ← Children ∪ {j};
(3b)  if (Children ∪ Unrelated) = (Neighbors/{parent}) then
(3c)      terminate.

(4)   When REJECT arrives from j:
(4a)  Unrelated ← Unrelated ∪ {j};
(4b)  if (Children ∪ Unrelated) = (Neighbors/{parent}) then
(4c)      terminate.
```

RAMAIAH
Institute of Technology

Asynchronous single-initiator spanning tree algorithm using flooding

- This algorithm assumes a designated root node which initiates the algorithm.
- The root initiates a flooding of QUERY messages in the graph to identify tree edges.
- The parent of a node is that node from which a QUERY is first received; an ACCEPT message is sent in response to such a QUERY.

```
(1)   When the predesignated root node wants to initiate the algorithm:
(1a)  if (i = root and parent = ⊥) then
(1b)      send QUERY to all neighbors;
(1c)      parent ⟵ i.

(2)   When QUERY arrives from j:
(2a)  if parent = ⊥ then
(2b)      parent ⟵ j;
(2c)      send ACCEPT to j;
(2d)      send QUERY to all neighbors except j;
(2e)      if (Children ∪ Unrelated) = (Neighbors/{parent}) then
(2f)          terminate.
(2g)  else send REJECT to j.

(3)   When ACCEPT arrives from j:
(3a)  Children ⟵ Children ∪ {j};
(3b)  if (Children ∪ Unrelated) = (Neighbors/{parent}) then
(3c)      terminate.

(4)   When REJECT arrives from j:
(4a)  Unrelated ⟵ Unrelated ∪ {j};
(4b)  if (Children ∪ Unrelated) = (Neighbors/{parent}) then
(4c)      terminate.
```

Asynchronous single-initiator spanning tree algorithm using flooding

- Other QUERY messages received are replied to by a REJECT message.
- Each node terminates its algorithm when it has received from all its non-parent neighbors a response to the QUERY sent to them.
- Procedures 1, 2, 3, and 4 are each executed atomically.

```
(1)   When the predesignated root node wants to initiate the algorithm:
(1a)  if (i = root and parent = ⊥) then
(1b)      send QUERY to all neighbors;
(1c)      parent ← i.

(2)   When QUERY arrives from j:
(2a)  if parent = ⊥ then
(2b)      parent ← j;
(2c)      send ACCEPT to j;
(2d)      send QUERY to all neighbors except j;
(2e)      if (Children ∪ Unrelated) = (Neighbors/{parent}) then
(2f)          terminate.
(2g)  else send REJECT to j.

(3)   When ACCEPT arrives from j:
(3a)  Children ← Children ∪ {j};
(3b)  if (Children ∪ Unrelated) = (Neighbors/{parent}) then
(3c)      terminate.

(4)   When REJECT arrives from j:
(4a)  Unrelated ← Unrelated ∪ {j};
(4b)  if (Children ∪ Unrelated) = (Neighbors/{parent}) then
(4c)      terminate.
```

RAMAIAH
Institute of Technology

Asynchronous single-initiator spanning tree algorithm using flooding

- Each node here needs to track its neighbors to determine which nodes are its children and which nodes are not.
- This tracking is necessary in order to know when to terminate.
- After sending QUERY messages on the outgoing links, the sender needs to know how long to keep waiting.
- This is accomplished by requiring each node to return an "acknowledgement" for each QUERY it receives.

```
(1)   When the predesignated root node wants to initiate the algorithm:
(1a)  if (i = root and parent = ⊥) then
(1b)      send QUERY to all neighbors;
(1c)      parent ⟵ i.

(2)   When QUERY arrives from j:
(2a)  if parent = ⊥ then
(2b)      parent ⟵ j;
(2c)      send ACCEPT to j;
(2d)      send QUERY to all neighbors except j;
(2e)      if (Children ∪ Unrelated) = (Neighbors/{parent}) then
(2f)          terminate.
(2g)  else send REJECT to j.

(3)   When ACCEPT arrives from j:
(3a)  Children ⟵ Children ∪ {j};
(3b)  if (Children ∪ Unrelated) = (Neighbors/{parent}) then
(3c)      terminate.

(4)   When REJECT arrives from j:
(4a)  Unrelated ⟵ Unrelated ∪ {j};
(4b)  if (Children ∪ Unrelated) = (Neighbors/{parent}) then
(4c)      terminate.
```

RAMAIAH Institute of Technology

Asynchronous single-initiator spanning tree algorithm using flooding

- The acknowledgement message has to be of a different type than the QUERY type.
- The algorithm in the figure uses two messages types besides the QUERY to distinguish between the child nodes and non-child nodes.
  1. ACCEPT (+ ack)
  2. REJECT (- ack)

```
(1)   When the predesignated root node wants to initiate the algorithm:
(1a)  if (i = root and parent = ⊥) then
(1b)      send QUERY to all neighbors;
(1c)      parent ⟵ i.

(2)   When QUERY arrives from j:
(2a)  if parent = ⊥ then
(2b)      parent ⟵ j;
(2c)      send ACCEPT to j;
(2d)      send QUERY to all neighbors except j;
(2e)      if (Children ∪ Unrelated) = (Neighbors/{parent}) then
(2f)          terminate.
(2g)  else send REJECT to j.

(3)   When ACCEPT arrives from j:
(3a)  Children ⟵ Children ∪ {j};
(3b)  if (Children ∪ Unrelated) = (Neighbors/{parent}) then
(3c)      terminate.

(4)   When REJECT arrives from j:
(4a)  Unrelated ⟵ Unrelated ∪ {j};
(4b)  if (Children ∪ Unrelated) = (Neighbors/{parent}) then
(4c)      terminate.
```

(1) When the predesignated root node wants to initiate the algorithm:
(1a) **if** ($i = root$ **and** $parent = \perp$) **then**
(1b) **send** QUERY to all neighbors;
(1c) $parent \longleftarrow i$.

(2) When QUERY arrives from j:
(2a) **if** $parent = \perp$ **then**
(2b) $parent \longleftarrow j$;
(2c) **send** ACCEPT to j;
(2d) **send** QUERY to all neighbors except j;
(2e) **if** $(Children \cup Unrelated) = (Neighbors/\{parent\})$ **then**
(2f) **terminate.**
(2g) **else send** REJECT to j.

(3) When ACCEPT arrives from j:
(3a) $Children \longleftarrow Children \cup \{j\}$;
(3b) **if** $(Children \cup Unrelated) = (Neighbors/\{parent\})$ **then**
(3c) **terminate.**

(4) When REJECT arrives from j:
(4a) $Unrelated \longleftarrow Unrelated \cup \{j\}$;
(4b) **if** $(Children \cup Unrelated) = (Neighbors/\{parent\})$ **then**
(4c) **terminate.**

RAMAIAH
Institute of Technology

Example :

1. A sends a QUERY to B and F.
2. F receives QUERY from A and determines that AF is a tree edge. F forwards the QUERY to E and C.
3. E receives a QUERY from F and determines that FE is a tree edge. E forwards the QUERY to B and D. C receives a QUERY from F and determines that FC is a tree edge. C forwards the QUERY to B and D.
4. B receives a QUERY from E and determines that EB is a tree edge. B forwards the QUERY to A, C, and D.
5. D receives a QUERY from E and determines that ED is a tree edge. D forwards the QUERY to B and C.

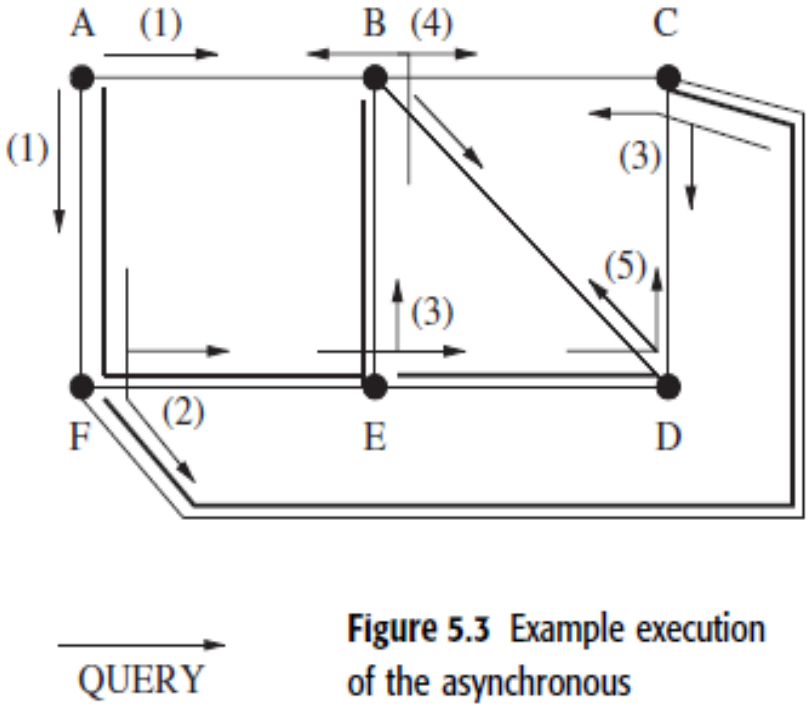


Figure 5.3 Example execution of the asynchronous flooding-based single initiator spanning tree algorithm (Algorithm 5.2).

RAMAIAH
Institute of Technology

Example :

- The resulting spanning tree rooted at A is shown in boldface.
- The numbers next to the QUERY messages indicate the approximate chronological order in which messages get sent.
- The same numbering used for messages sent by different nodes implies that those actions occur concurrently and independently.

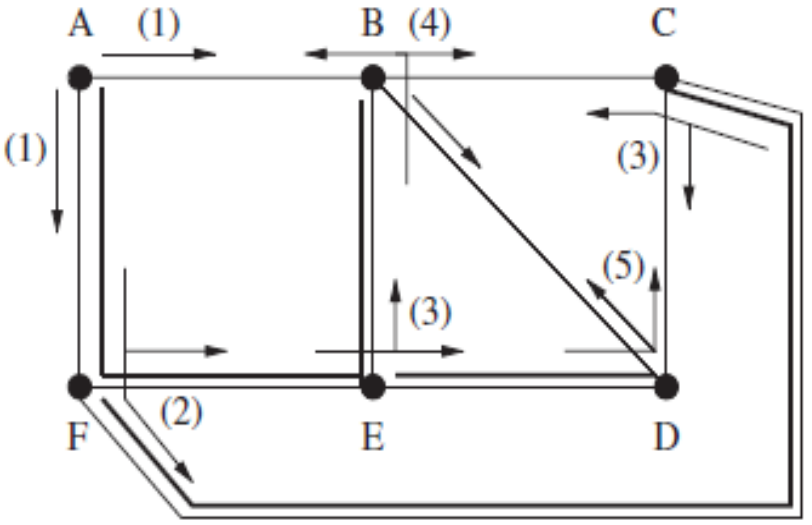


Figure 5.3 Example execution of the asynchronous flooding-based single initiator spanning tree algorithm (Algorithm 5.2).

Synchronizers

It is much more difficult to design the algorithm for an asynchronous system, than for a synchronous system.

A general technique to convert an algorithm designed for a synchronous system, to run on an asynchronous system.

The generic class of transformation algorithms to run synchronous algorithms on asynchronous systems are called synchronizers.

Synchronizers

Formally, a synchronizer sits between the underlying network and the processes and does one of two things:

A **global synchronizer** guarantees that no process receives a message for round r until *all processes* have sent their messages for round r.

A **local synchronizer** guarantees that no process receives a message for round r until *all of that process's neighbors* have sent their messages for round r.

Synchronizers

Definition

Class of transformation algorithms that allow a synchronous program (designed for a synchronous system) to run on asynchronous systems.

- Assumption: failure-free system
- Designing tailor-made async algo from scratch may be more efficient than using synchronizer

Process safety

Process i is safe in round r if all messages sent by i have been received.

Implementation key: signal to each process when it is safe to go to next round, i.e., when all msgs to be received have arrived

Synchronizers

The mesage complexity Ma and time complexity Ta of the asynchronous algorithm are as follows:

$$M_a = M_s + (M_{init} + rounds \cdot M_{round})$$
$$T_a = T_s + T_{init} + rounds \cdot T_{round}$$

- M_s: # messages in the synchronous algorithm.
- *rounds*: # rounds in the synchronous algorithm.
- T_s: time for the synchronous algorithm.
  Assuming one unit (message hop) per round, this equals *rounds*.
- M_{round}: # messages needed to simulate a round,
- T_{round}: # sequential message hops to simulate a round.
- M_{init}, T_{init}: # messages, # sequential message hops to initialize async system.

Synchronizers

Table 5.1 The message and time complexities for the *simple*, α, β, and γ synchronizers. h_c is the greatest height of a tree among all the clusters. L_c is the number of tree edges and designated edges in the clustering scheme for the γ synchronizer. d is the graph diameter.

	Simple synchronizer	α synchronizer	β synchronizer	γ synchronizer
M_{init}	0	0	$O(n \cdot log(n) + \|L\|)$	$O(kn^2)$
T_{init}	d	0	$O(n)$	$n \cdot log(n)/log(k)$
M_{round}	$2\|L\|$	$O(\|L\|)$	$O(n)$	$O(L_c)$ $(\leq O(kn))$
T_{round}	1	$O(1)$	$O(n)$	$O(h_c)$ $(\leq O(log(n)/log(k)))$

Synchronizers

Four standard synchronizers proposed by Awerbuch

1. The simple
2. α
3. β
4. **γ**

The α, β, **γ** synchronizers use the notion of process safety, defined as follows.

The α and β synchronizers are extreme cases of the **γ** synchronizer and form its building blocks.

The main difference between them is the mechanism used to determine when round-r messages have been delivered.

A simple synchronizer

- A process sends each neighbor 1 message/round
  Combine messages or send dummy message
- On receiving a msg from each neighbor, go to next round.
- Neighbors P_i, P_j may be only one round apart
- P_i in $round_i$ can receive msg from only $round_i$ or $round_i + 1$ of neighbor.
- Initialization:
  - Any process may start round x.
  - In d time units, all processes would be in round x.
  - $T_{init} = d, M_{init} = 0$.
- Complexity: $M_{round} = 2|L|, T_{round} = 1$.

α synchronizer

- In the alpha synchronizer, every node sends a message to every neighbor in every round (possibly a dummy message if the underlying protocol doesn't send a message);
- this allows the receiver to detect when it's gotten all its round-r messages (because it expects to get a message from every neighbor)

α synchronizer

The operation is illustrated in Figure 5.10.

1. (step 1) Node A sends a message to nodes C and E, and receives messages from B and E in the same round.
2. (step 2) These messages are acknowledged after they are received.

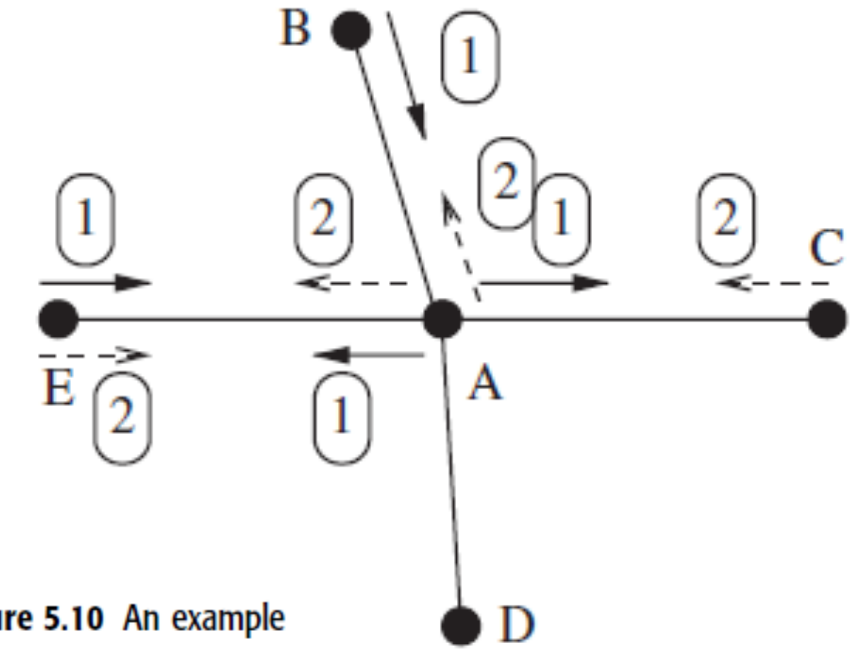
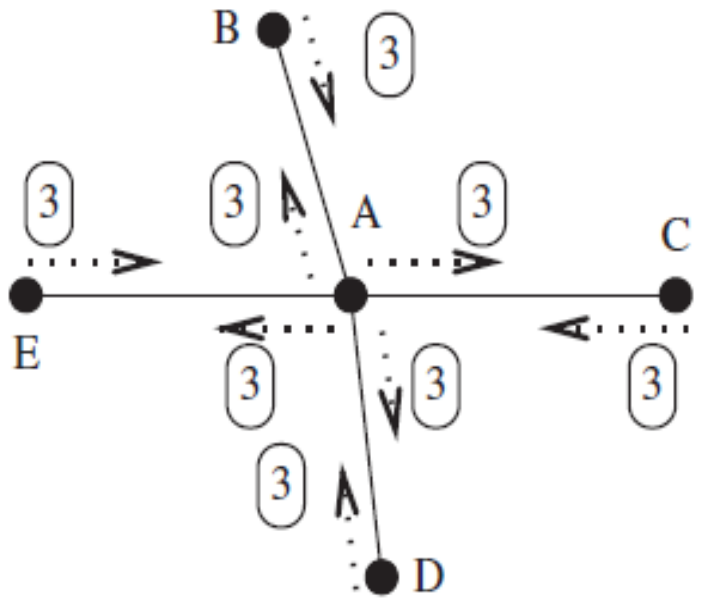


Figure 5.10 An example showing steps of the α synchronizer. (a) Execution messages (step 1) and their acknowledgements (step 2).

α synchronizer

The operation is illustrated in Figure 5.10.

3. (step 3) Once node A receives the acknowledgements from C and E, it sends a message to all its neighbors to notify them that node A is safe.



(b) "I am safe" messages (step 3).

α synchronizer

- Complexity:
  - l' msgs $\Rightarrow$ l' acks; transport layer acks $\Rightarrow$free!
  - $2|L|$ messages/round to inform neighbors of safety.
    $M_{round} = O(|L|).T_{round} = O(1)$.
- Initialization: None. Any process may spontaneously wake up.

β synchronizer

- In the beta synchronizer, messages are acknowledged by their receivers, so the senders can detect when all of their messages are delivered.

Initialization: rooted spanning tree, $O(n \log n + |L|)$ messages, $O(n)$ time.
Operation:

- Safe nodes initiate convergecast (CvgC)
- intermediate nodes propagate CvgC when their subtree is safe.
- When root becomes safe and receives CvgC from all children, initiates tree broadcast to inform all to move to next round.

Complexity: l' acks for free, due to transport layer.

- $M_{round} = 2(n-1)$
- $T_{round} = 2 \log n$ average; $2n$ worst case

γ synchronizer

The network is organized into a set of clusters, as shown in Figure 5.11.

Within a cluster, a spanning tree hierarchy exists with a distinguished root node.

The height of a clustering scheme, hc, is the maximum height of the spanning trees across all of the clusters.

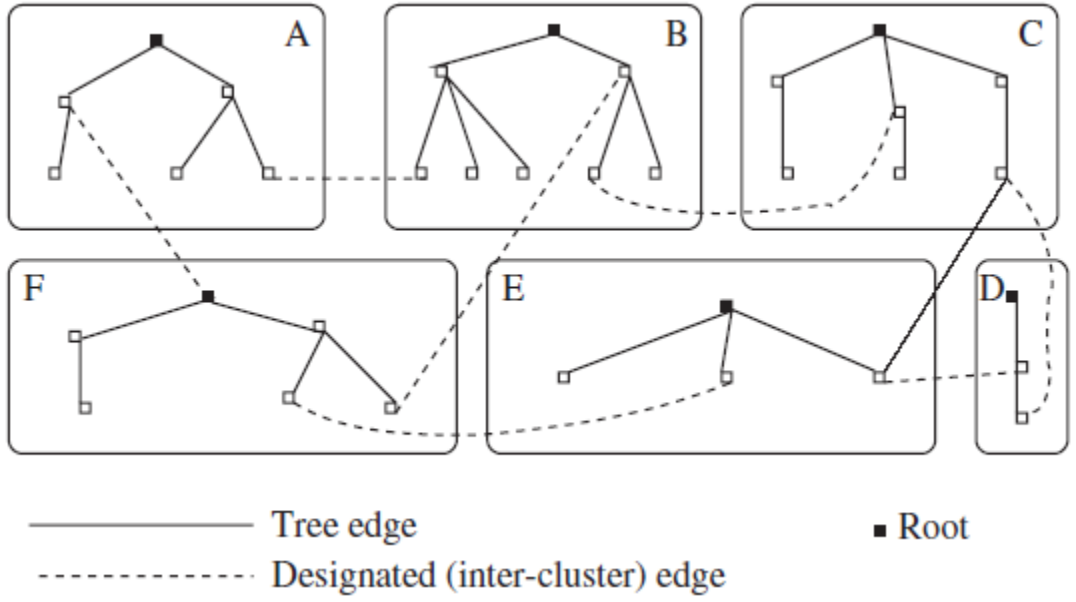


Figure 5.11 Cluster organization for the γ synchronizer, showing six clusters A–F. Only the tree edges within each cluster, and the inter-cluster *designated* edges are shown.

γ synchronizer

Two clusters are neighbors if there is at least one edge between one node in each of the two clusters; one of such multiple edges is the designated edge for that pair of clusters.

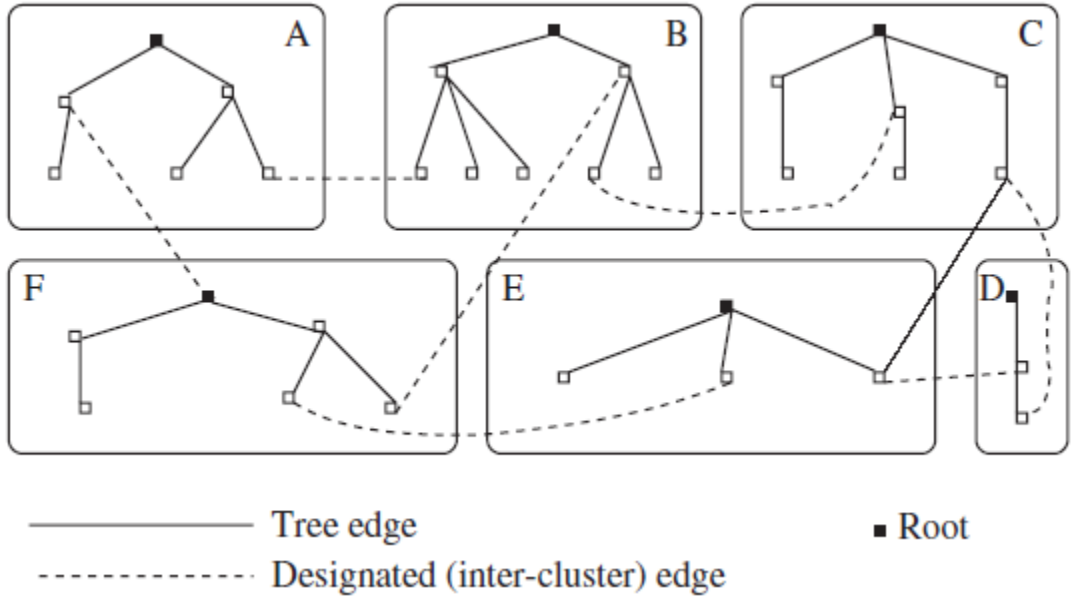


Figure 5.11 Cluster organization for the γ synchronizer, showing six clusters A–F. Only the tree edges within each cluster, and the inter-cluster *designated* edges are shown.

γ synchronizer

Within a cluster, the β synchronizer is executed; once a cluster is "stabilized,"

the **α** synchronizer is executed among the clusters, over the designated edges.

To convey the results of the stabilization of the inter-cluster **α** synchronizer, within each cluster, a convergecast and broadcast phase is then executed.

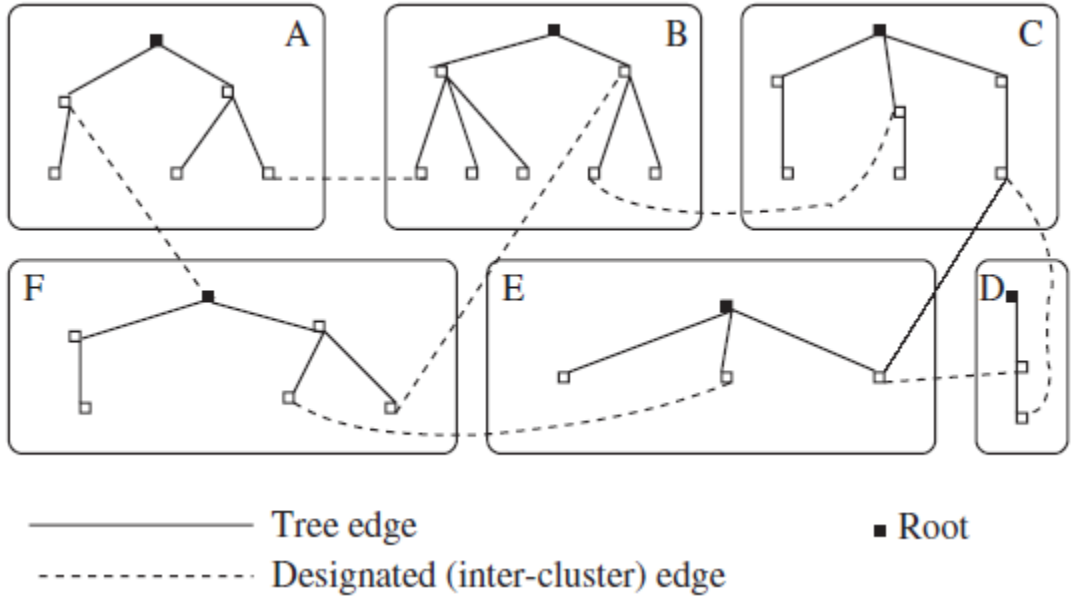


Figure 5.11 Cluster organization for the γ synchronizer, showing six clusters A–F. Only the tree edges within each cluster, and the inter-cluster *designated* edges are shown.

γ synchronizer

Over the designated intercluster edges, two types of messages are exchanged for the synchronizer:

1. My_cluster_safe,
2. Neighboring_cluster_safe,

with semantics that are self evident.

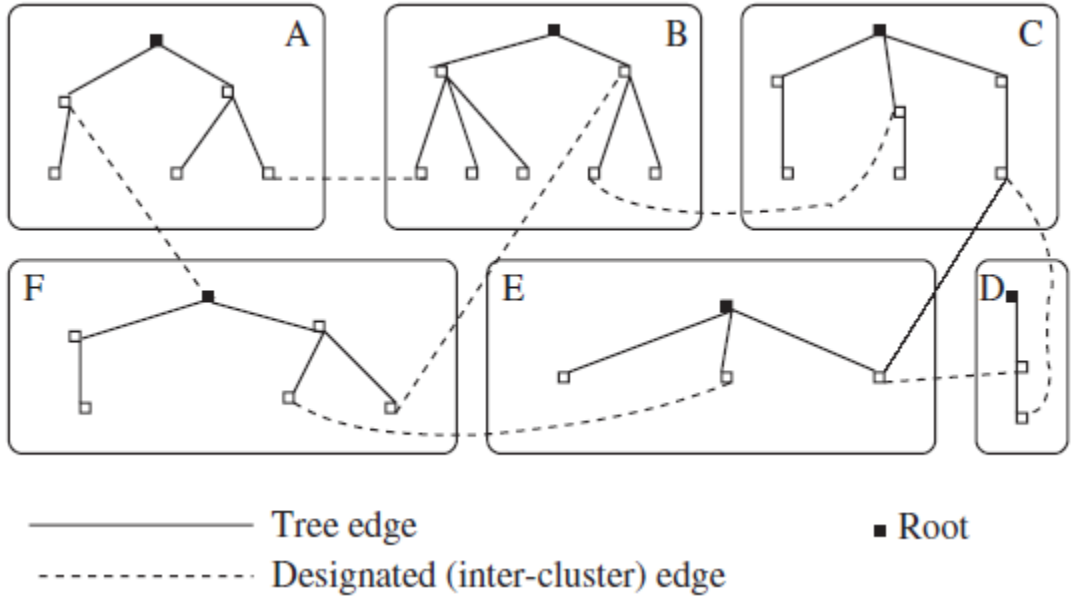


Figure 5.11 Cluster organization for the γ synchronizer, showing six clusters A–F. Only the tree edges within each cluster, and the inter-cluster *designated* edges are shown.

Maximal independent set (MIS)

A MIS is also a dominating set in the graph, and **every dominating set that is independent must be maximal independent**, so MISs are also called independent dominating sets.

Given a graph G = (V, E), determine an independent I ⊆ V of largest possible size.

Maximal independent set (MIS)

- For a graph (N, L), an *independent set* of nodes N', where $N' \subset N$, is such that for each i and j in N', $(i, j) \notin L$.
- An independent set N' is a *maximal independent set* if no strict superset of N' is an independent set.
- A graph may have multiple MIS; perhaps of varying sizes.
  The largest sized independent set is the *maximum independent set*.
- Application: wireless broadcast - allocation of frequency bands (mutex)
- NP-complete

Luby's Randomized Algorithm, Async System

Iteratively:

- Nodes pick random nos, exchange with nbhs
- Lowest number in neighborhood wins (selected in MIS)
- If neighbor is selected, I am eliminated ($\Rightarrow$ safety)
- Only neighbors of selected nodes are eliminated ($\Rightarrow$ correctness)

Complexity:

- In each iteration, ≥ 1 selected, ≥ 1 eliminated $\Rightarrow \leq n/2$ iterations.
- Expected # iterations $O(log, n)$ due to randomized nature.

Luby's Randomized Algorithm, Async System

```
(variables)
set of integer Neighbours                                   // set of neighbours
real random_i                                               // random number from a sufficiently large range
boolean selected_i                                          // becomes true when P_i is included in the MIS
boolean eliminated_i                                        // becomes true when P_i is eliminated from the candidate set
(message types)
RANDOM(real random)                                         // a random number is sent
SELECTED(integer pid, boolean indicator)                    // whether sender was selected in MIS
ELIMINATED(integer pid, boolean indicator)                  // whether sender was removed from candidates

(1a) repeat
(1b) if Neighbours = ∅ then
(1c)       selected_i ⟵ true; exit();
(1d) random_i ⟵ a random number;
(1e) send RANDOM(random_i) to each neighbour;
(1f) await RANDOM(random_j) from each neighbour j ∈ Neighbours;
(1g) if random_i < random_j (∀j ∈ Neighbours) then
(1h)       send SELECTED(i, true) to each j ∈ Neighbours;
(1i)       selected_i ⟵ true; exit();  // in MIS
(1j) else
(1k)       send SELECTED(i, false) to each j ∈ Neighbours;
(1l)       await SELECTED(j, ⋆) from each j ∈ Neighbours;
(1m)       if SELECTED(j, true) arrived from some j ∈ Neighbours then
(1n)            for each j ∈ Neighbours from which SELECTED(⋆, false) arrived do
(1o)                 send SELECTED(i, true) to j;
(1p)            eliminated_i ⟵ true; exit();  // not in MIS
(1q)       else
(1r)            send ELIMINATED(i, false) to each j ∈ Neighbours;
(1s)            await ELIMINATED(j, ⋆) from each j ∈ Neighbours;
(1t)            for all j ∈ Neighbours do
(1u)                 if ELIMINATED(j, true) arrived then
(1v)                      Neighbours ⟵ Neighbours \ {j};
(1w) forever.
```

Luby's Randomized Algorithm, Async System

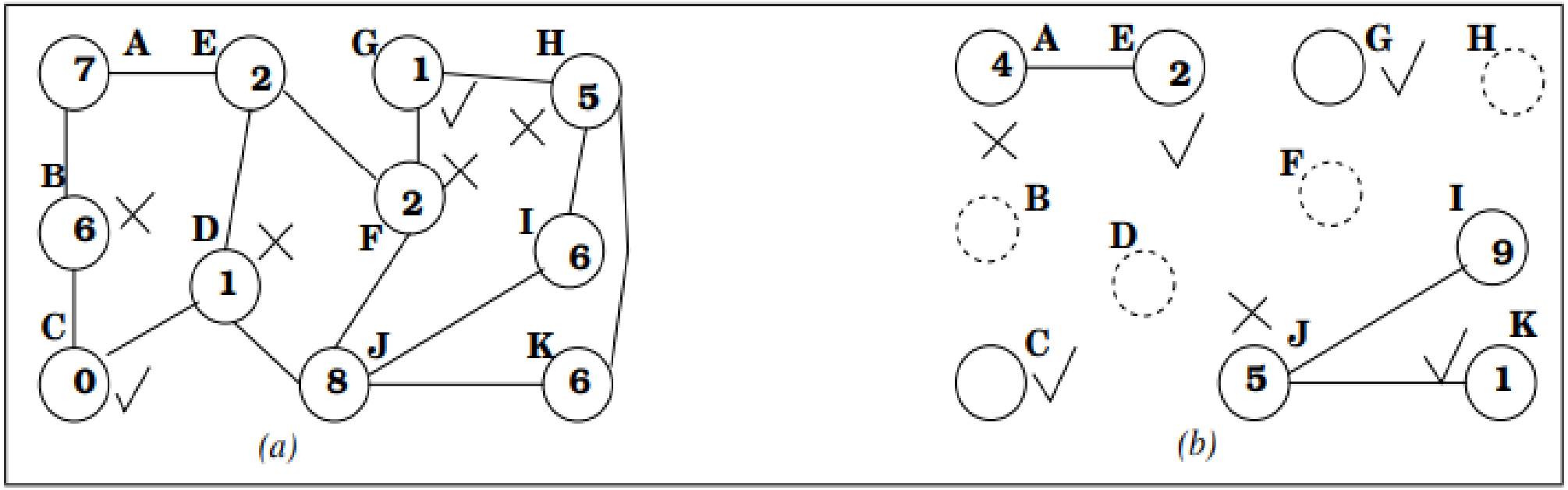


Figure 5.12: (a) Winners and losers in round 1. (b) Winners up to round 1, losers in round 2.

Third round: I is winner. MIS=$\{C, E, G, I, K\}$.

Leader election

- Defn: All processes agree on a common distinguished process (leader)
- Distributed algorithms not completely symmetrical; need a initiator, finisher process; e.g., MST for BC and CvgC to compute global function
- LeLang Chang Roberts (LCR) algorithm
  - Asynchronous unidirectional ring
  - All processes have unique IDs
  - Processes circulate their IDs; highest ID wins
  - Despite obvious optimizations, msg complexity $n \cdot (n-1)/2$; time complexity $O(n)$.
- Cannot exist deterministic leader election algorithm for anonymous rings
- Algorithms may be uniform

Leader election

- Many distributed algorithms need one process to act as coordinator
  - Doesn't matter which process does the job, just need to pick one
- Election algorithms: technique to pick a unique coordinator (aka *leader election*)
- Types of election algorithms: Bully and Ring algorithms

Leader election

- Processes have unique Ids and arranged in a logical ring
- Each process knows its neighbors
- Select process with highest ID as leader
- Begin election if just recovered or coordinator has failed
- Send *Election* to closest downstream node that is alive
  - Sequentially poll each successor until a live node is found
- Each process tags its ID on the message
- Initiator picks node with highest ID and sends a coordinator message
- Multiple elections can be in progress—no harm.

Leader election

- Defn: All processes agree on a common distinguished process (leader)
- Distributed algorithms not completely symmetrical; need a initiator, finisher process; e.g., MST for BC and CvgC to compute global function
- LeLang Chang Roberts (LCR) algorithm
  - Asynchronous unidirectional ring
  - All processes have unique IDs
  - Processes circulate their IDs; highest ID wins
  - Despite obvious optimizations, msg complexity $n \cdot (n-1)/2$; time complexity $O(n)$.
- Cannot exist deterministic leader election algorithm for anonymous rings
- Algorithms may be uniform

Leader election

```
(variables)
boolean participate ← false          // becomes true when P_i is included in the MIS
(message types)
PROBE integer                        // contains a node identifier
SELECTED integer                     // announcing the result

(1) When a process wakes up to participate in leader election:
(1a) send PROBE(i) to right neighbor;
(1b) participate ⟵ true.

(2) When a PROBE(k) message arrives from the left neighbor P_j:
(2a) if participate = false then execute step (1) first.
(2b) if i > k then
(2c)          discard the probe;
(2d) else if i < k then
(2e)          forward PROBE(k) to right neighbor;
(2f) else if i = k then
(2g)          declare i is the leader;
(2h)          circulate SELECTED(i) to right neighbor;

(3) When a SELECTED(x) message arrives from left neighbor:
(3a) if x ≠ i then
(3b)          note x as the leader and forward message to right neighbor;
(3c) else do not forward the SELECTED message.
```

Leader election

- Principle :

➢Process priority is obtained by organizing processes into a (logical) ring. Process with the highest priority should be elected as coordinator.

❑When a process notices that co-ordinator is not functioning :

➢Builds an ELECTION message (containing its own process number)

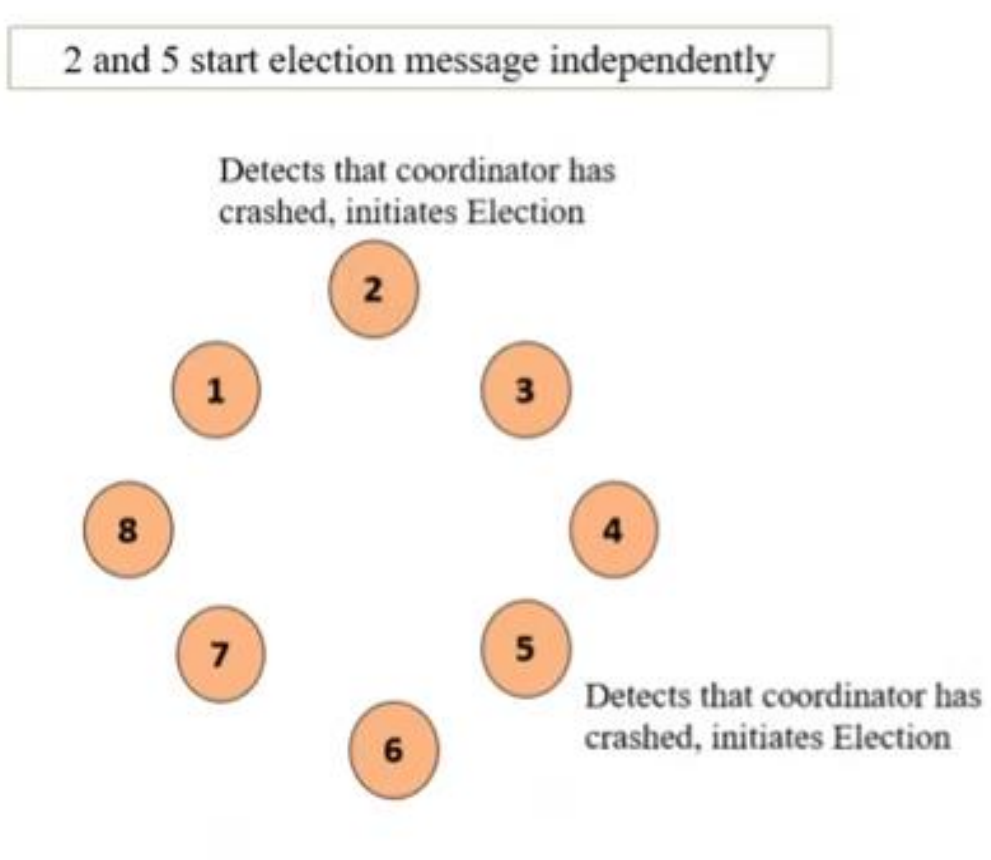
➢Sends the message to its successor (if successor is down, sender skips over it and goes to the next member along the ring, or the one after that, until a running process is located.

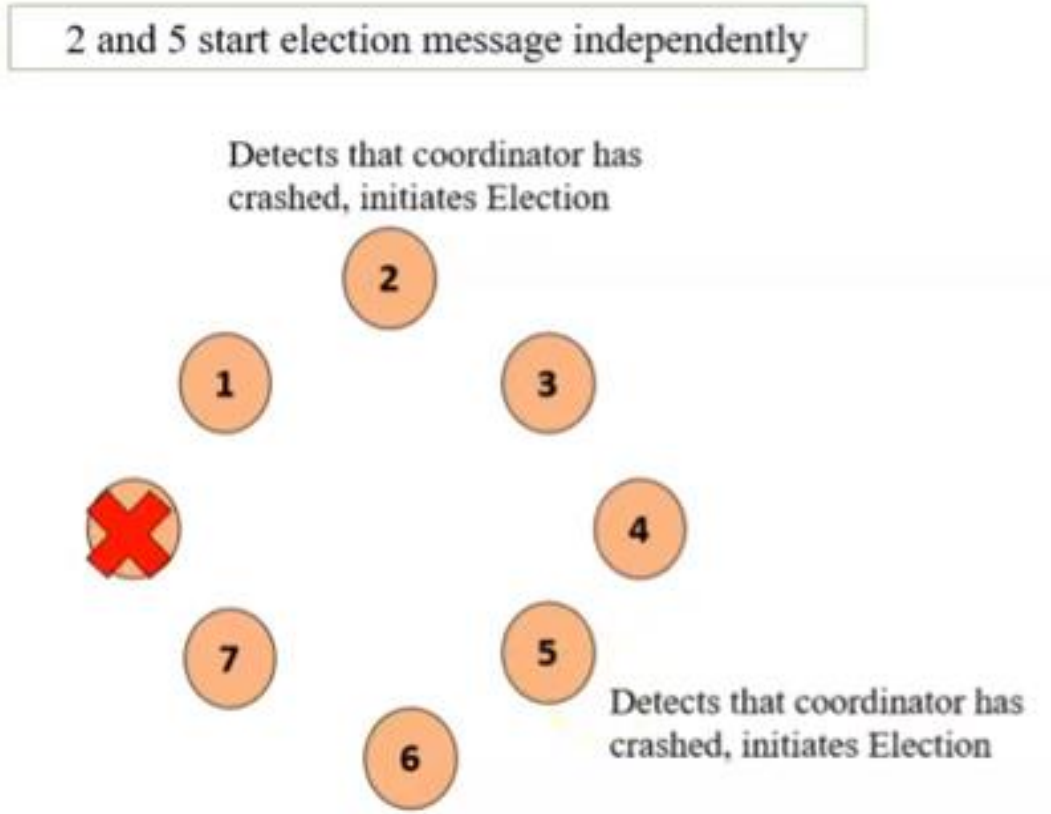
➢At each step, sender adds its own process number to the list in the message.

Leader election

- When the message gets back to the process that started it all :
  - Message comes back to the initiator.
  - In the queue, the process with maximum process ID wins.
  - Initiator announces the winner by sending another message around the ring.

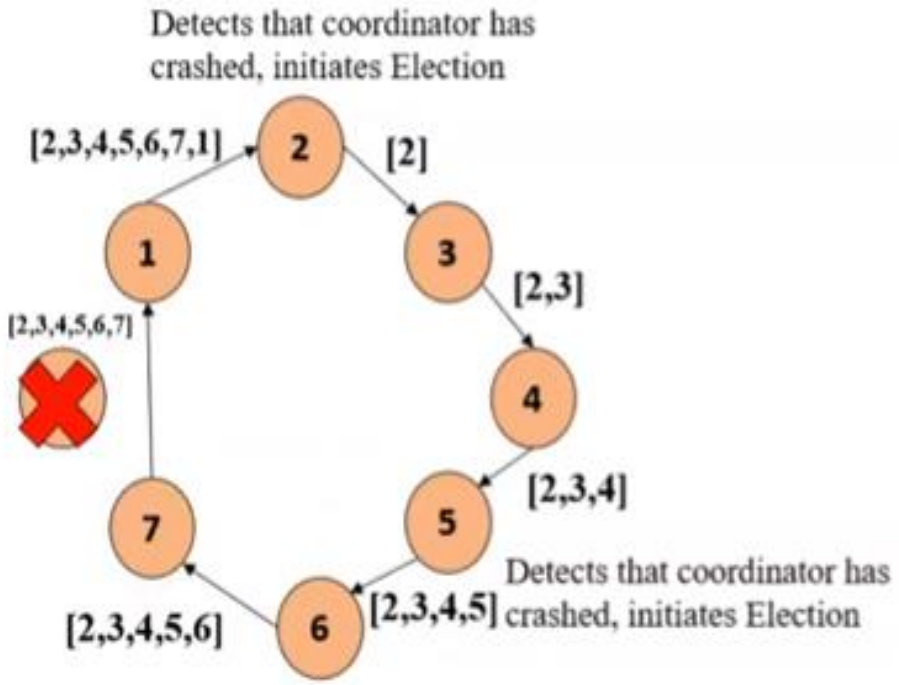
Example


2 and 5 start election message independently
Detects that coordinator has crashed, initiates Election
2
1
3
8
4
7
5
6
Detects that coordinator has crashed, initiates Election

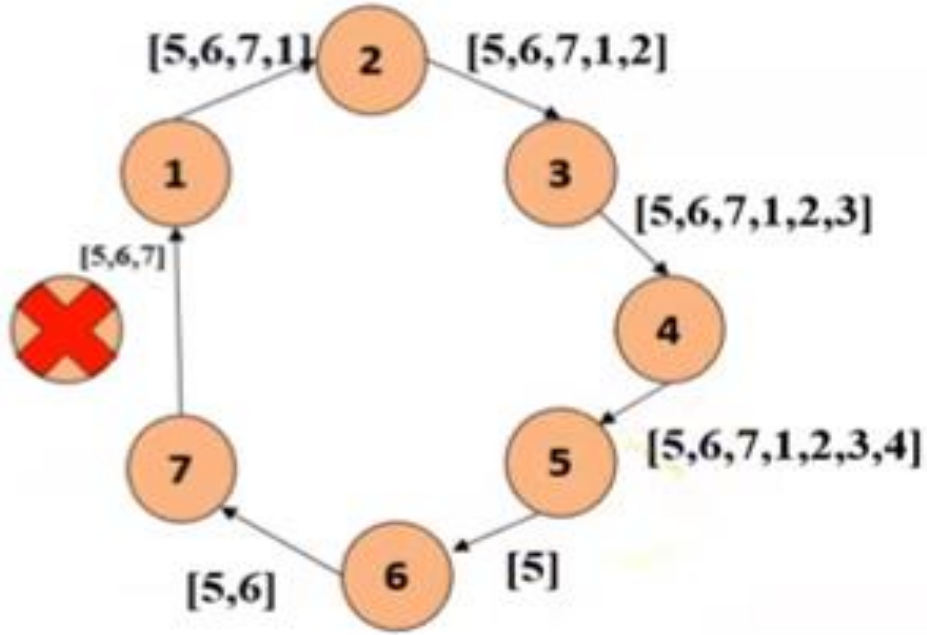

2 and 5 start election message independently
Detects that coordinator has crashed, initiates Election
2
1
3
4
7
5
6
Detects that coordinator has crashed, initiates Election

Example

2 and 5 start election message independently



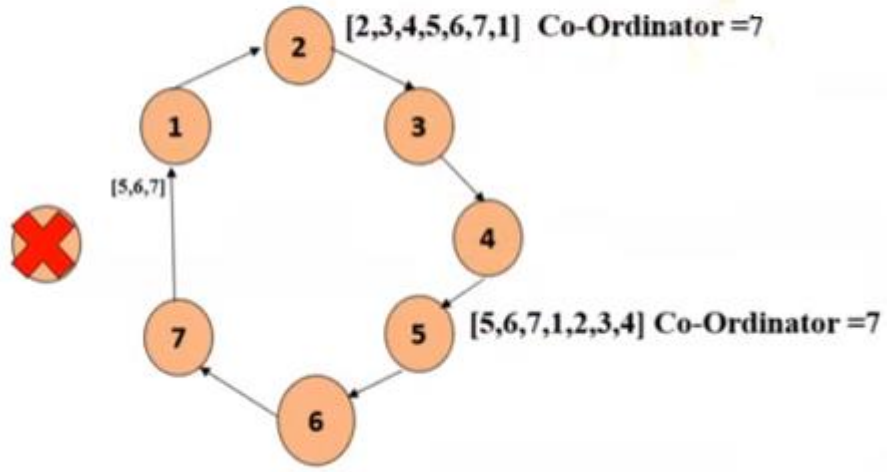
Both messages continue to circulate.



Example

https://www.youtube.com/watch?v=TzwiGTbUSHg

2 and 5 will convert Election messages to COORDINATOR messages.
All processes recognize highest numbered process as new coordinator.



THANK YOU