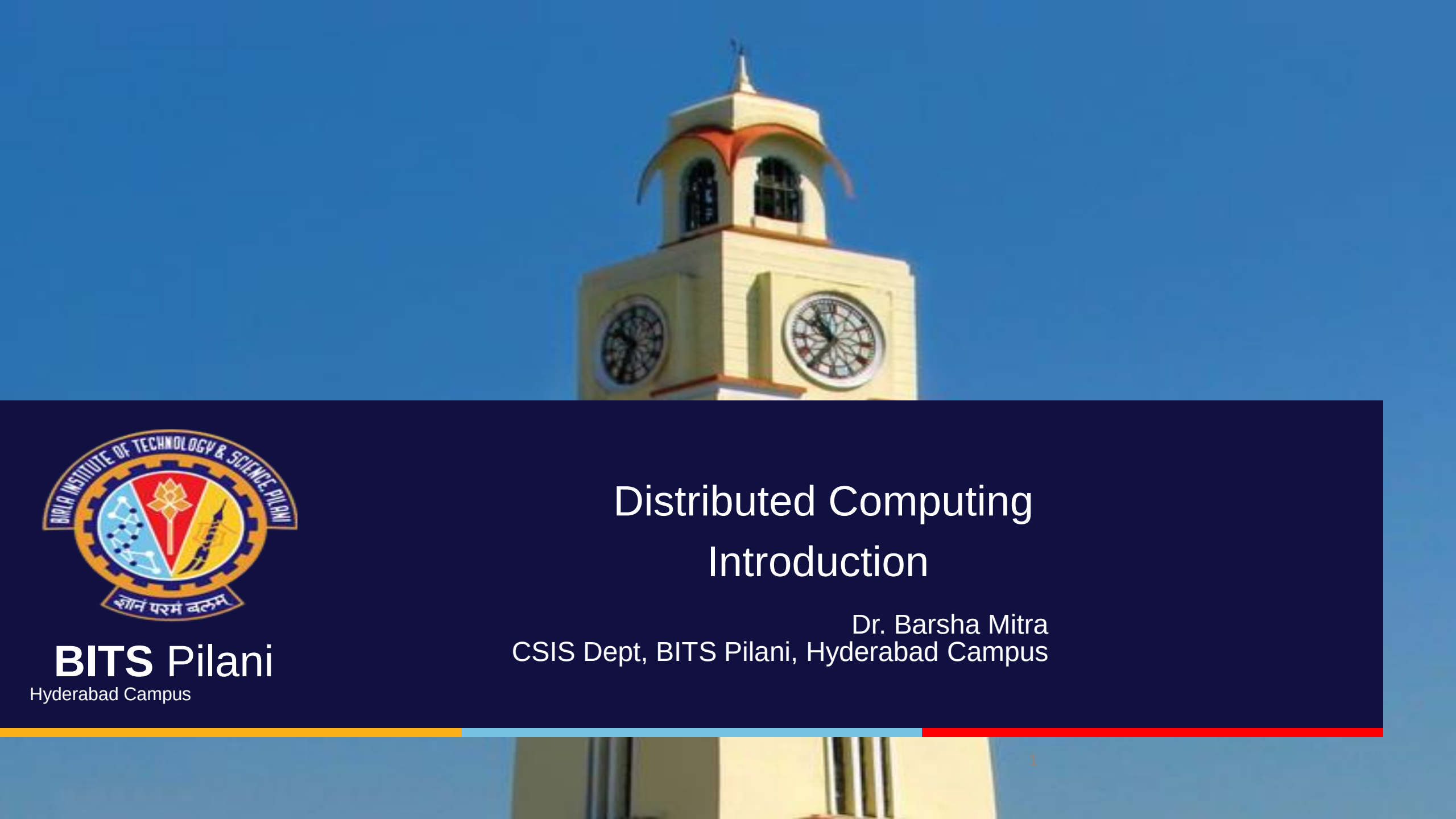


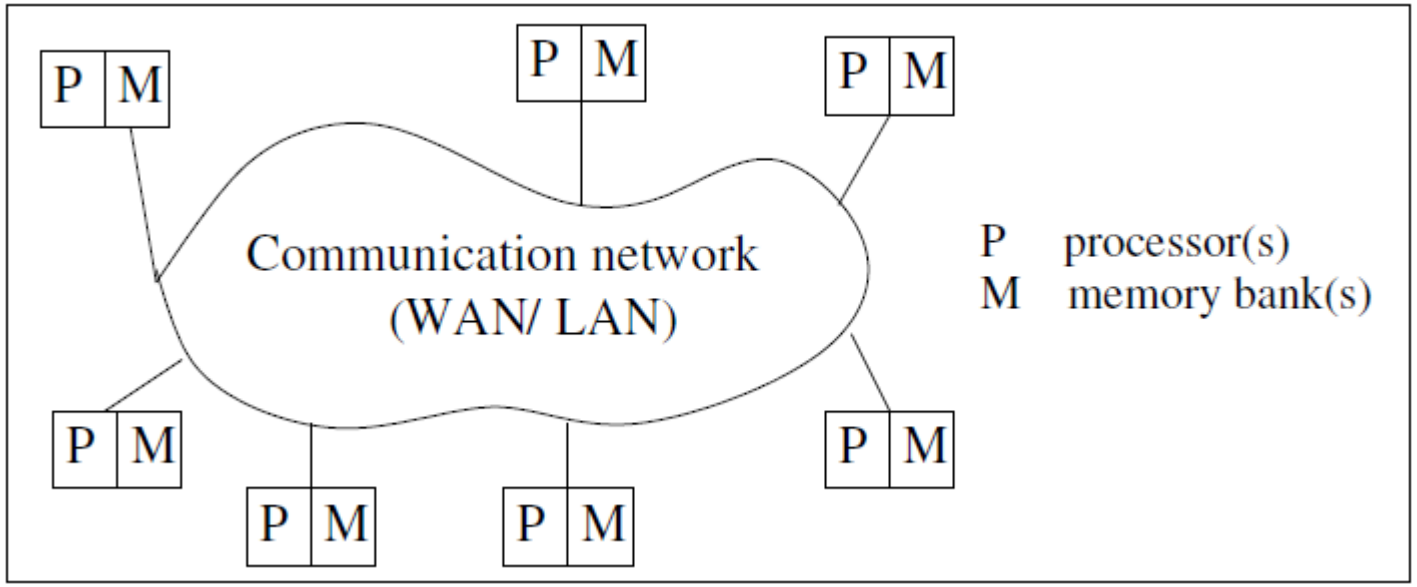
Distributed Computing
Introduction

Dr. Barsha Mitra
CSIS Dept, BITS Pilani, Hyderabad Campus

BITS Pilani
Hyderabad Campus

Definition of Distributed Systems

- Collection of independent entities cooperating to collectively solve a task
- Autonomous processors communicating over a communication network

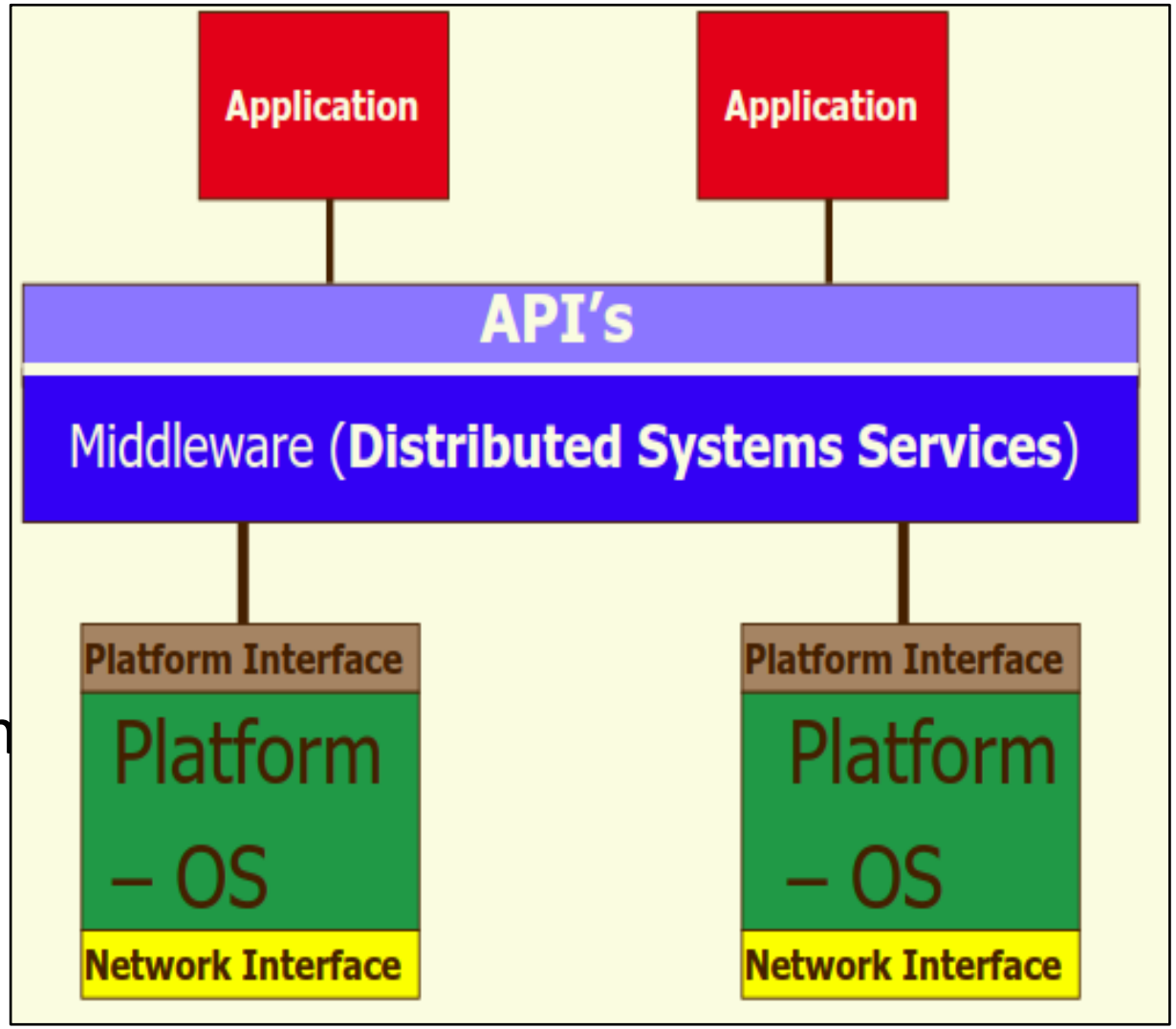


Features of Distributed System

- **No common physical clock**
- **No shared memory** - Requires message passing for communication
- **Geographical seperation**
  - Processors are geographically wide apart
  - Processors may reside on a WAN or LAN
- **Autonomy and Heterogeneity**
  - Loosely coupled processors
  - Different processor speeds and operating systems
  - Processors are not part of a dedicated system
  - Cooperate with one another
  - May offer services or solve a problem jointly

Middleware

- **Middleware** –
  - distributed software
  - drives the distributed system
  - provides transparency of heterogeneity at platform level
- **Middleware standards**:
  - Common Object Request Broker Architecture (CORBA)
  - Remote Procedure Call (RPC) mechanism
  - Distributed Component Object Model (DCOM)
  - Remote Method Invocation (RMI)



Motivation for Distributed System

- Inherently distributed computation
- Resource sharing
- Access to remote resources
- Increased performance/cost ratio
- Reliability
- Scalability
- Modularity and Incremental Expandability

Coupling

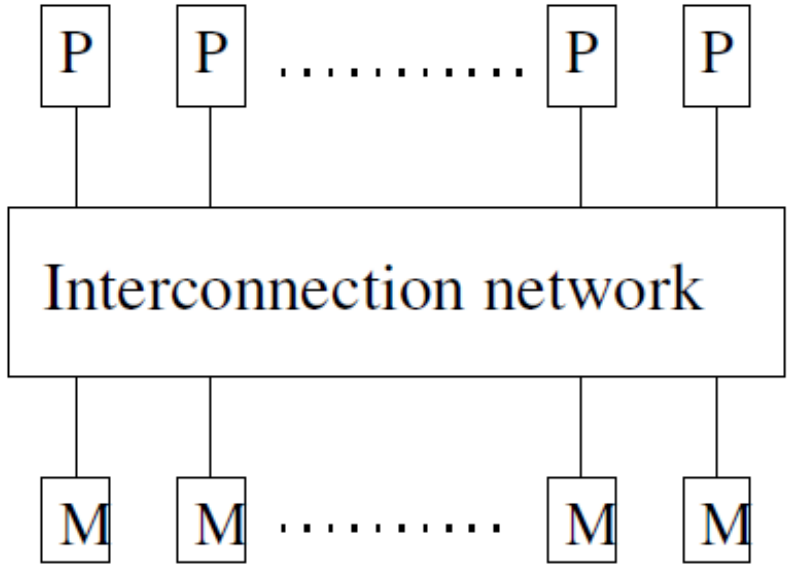
- Interdependency/binding and/or homogeneity among modules, whether hardware or software (e.g., OS, middleware)
- High coupling → tightly coupled modules
- Low coupling → loosely coupled modules

Parallel Systems

- **Multiprocessor systems**
  - E.g., Omega, Butterfly Networks
- **Multicomputer parallel systems**
  - E.g., NYU Ultracomputer, CM* Connection Machine, IBM Blue gene
- **Array processors** (collocated, tightly coupled, common system clock, may not have shared memory)
  - E.g., DSP applications, image processing

UMA Model

- Direct access to shared memory
- **Access latency** - waiting time to complete an access to any memory location from any processor
- Access latency is same for all processors
- Processors
  - Remain in close proximity
  - Connected by an interconnection network

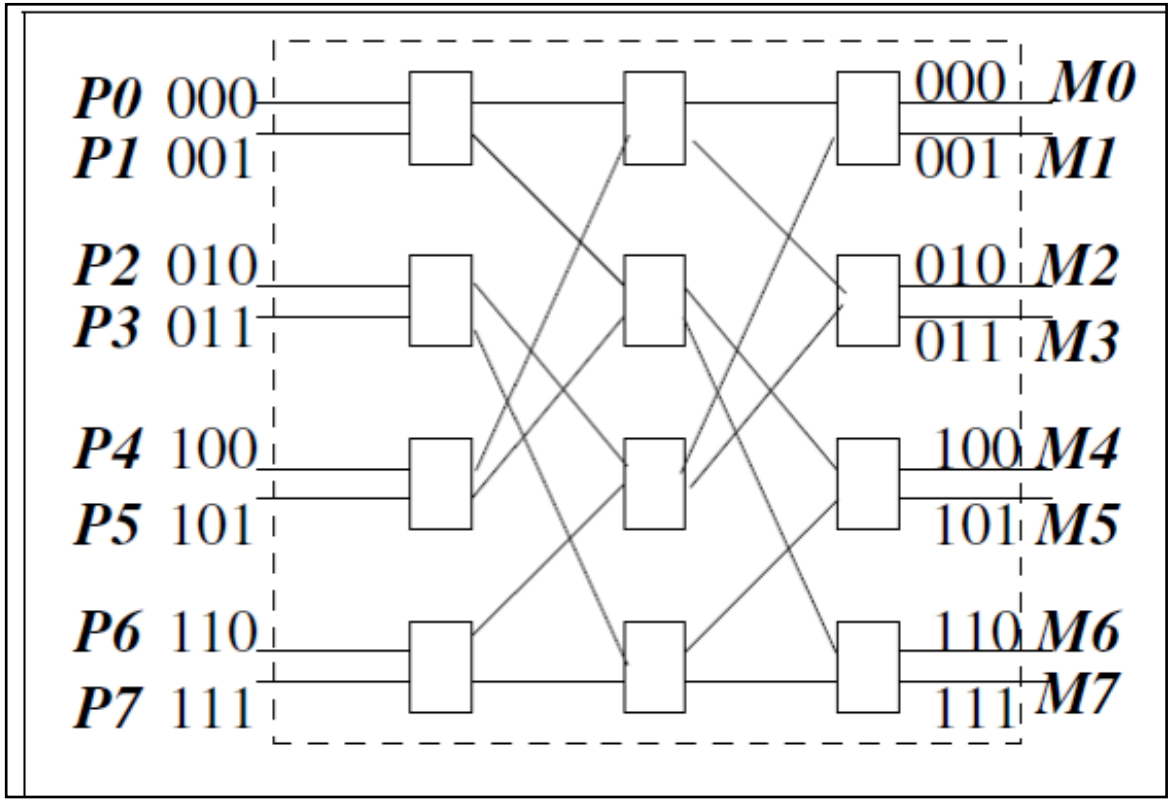


UMA Model

- Interprocess communication occurs through read and write operations on shared memory
- Processors are of same type
- Remain within same physical box/container
- Processors run the same operating system
- Hardware & software are very tightly coupled
- Interconnection network to access memory may be
    - Bus
    - Multistage switch

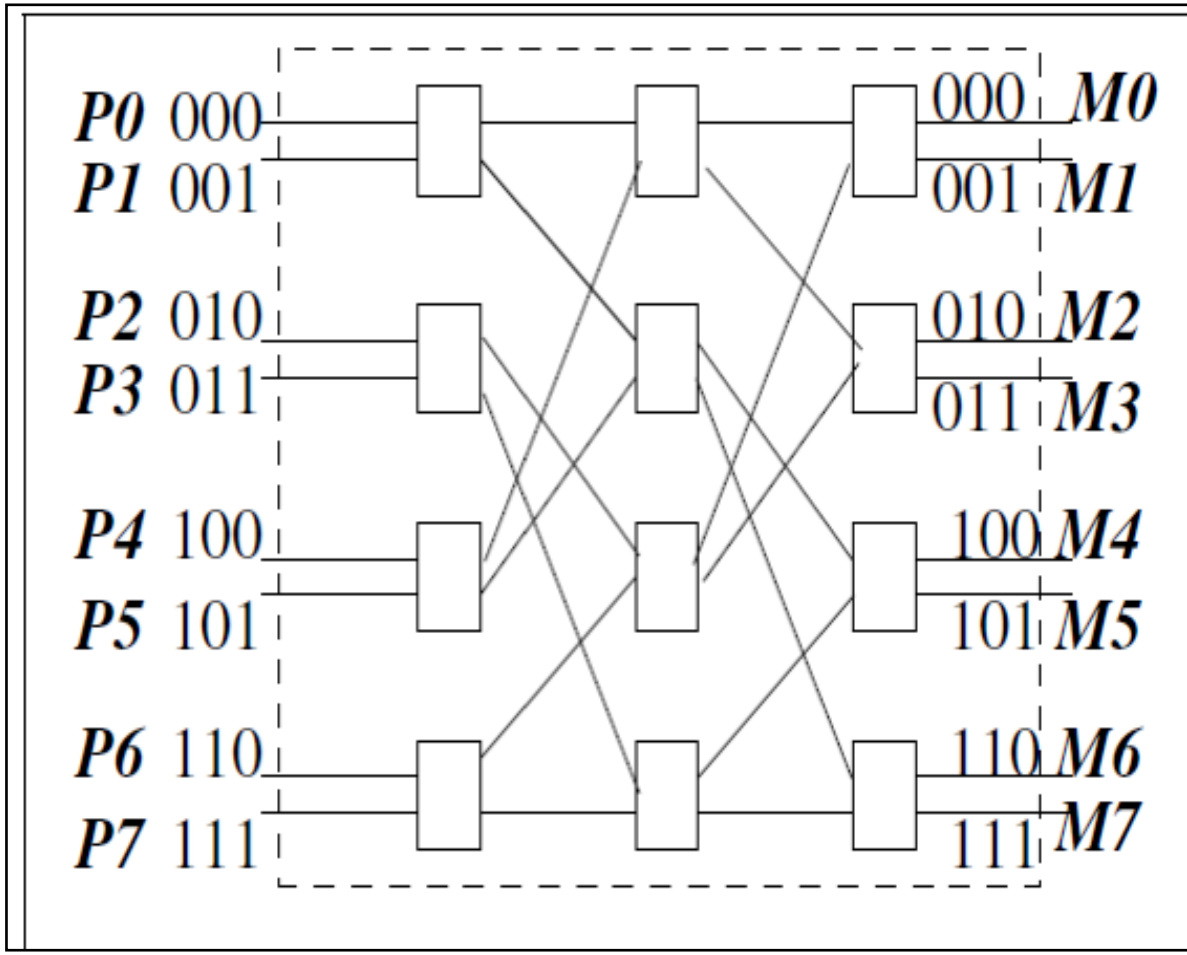
Omega Network

- Multi-stage networks formed by 2 x 2 switching elements
- Data can be sent on any one of the input wires
- n-input and n-output network uses:
  - $\log_2(n)$ stages
  - $\log_2(n)$ bits for addressing
- n processors, n memory banks



Omega Network

- $(n/2)\log_2(n)$ switching elements of size 2x2
- **Interconnection function**: Output i of a stage is connected to input j of next stage:
  - **$j = 2i$, for $0 \le i \le n/2 - 1$**
    $= 2i + 1 - n$, for $n/2 \le i \le n - 1$
  - left-rotation operation on the binary representation of i to get j



Omega Network

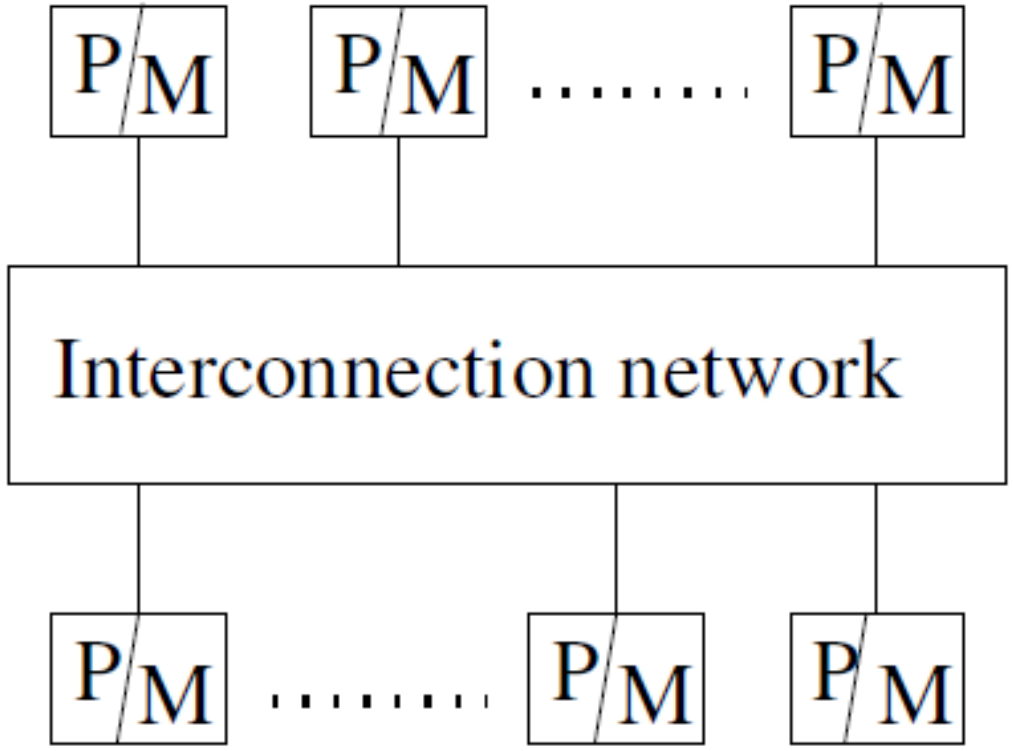
- **Routing function**: from input line i to output line j
- considers j and stage number s, s ϵ [0, $\log_2(n) - 1$]
- in a stage s switch
  - if $(s + 1)^{th}$ MSB of j is 0, then route on upper wire
  - else [i.e., $(s + 1)^{th}$ MSB of j is 1], route on lower wire

Multicomputer Parallel Systems

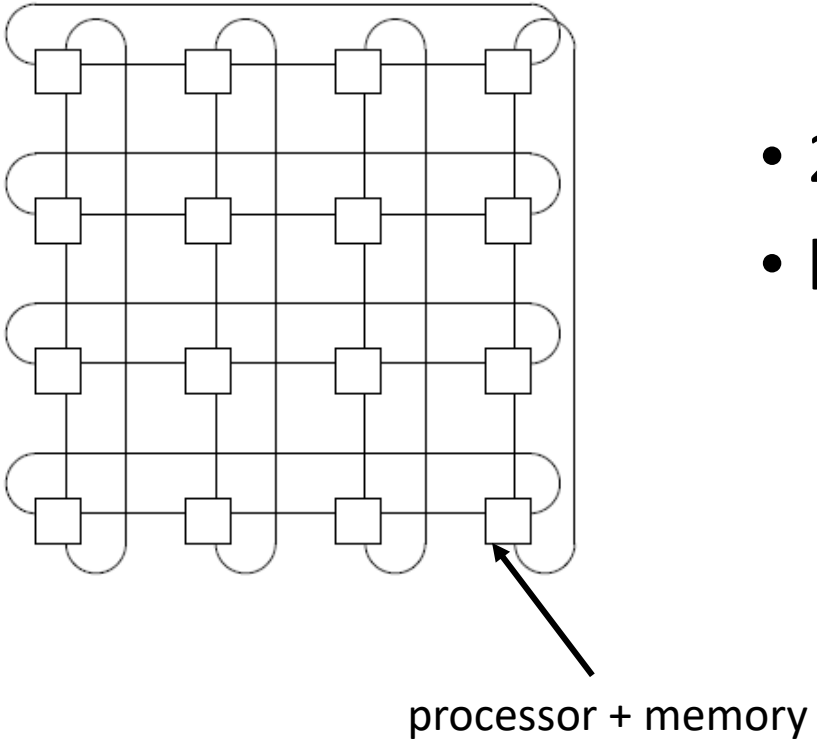
- multiple processors *do not have direct access to shared memory*
- usually do not have a common clock
- processors are in close physical proximity
- very tightly coupled (homogenous hardware and software)
- connected by an interconnection network
- communicate either via a common address space or via message-passing

NUMA Model

- non-uniform memory access architecture
- multicomputer system having a common address space
- latency to access various shared memory locations from the different processors varies

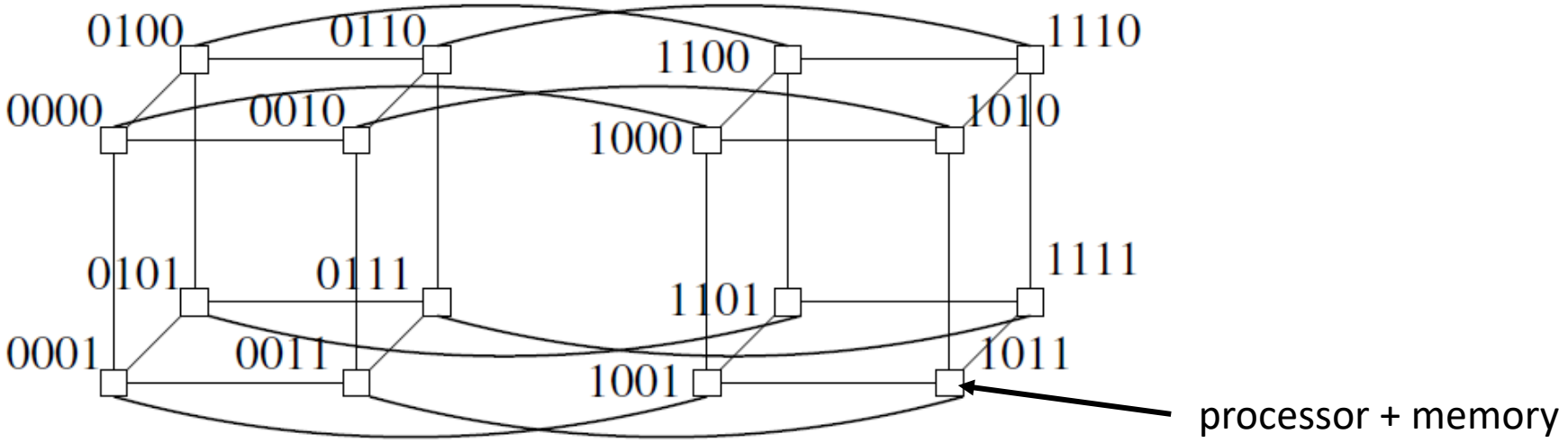


Multicomputer Parallel Systems



- 2-D wraparound mesh
- k x k mesh contains k^2 processors

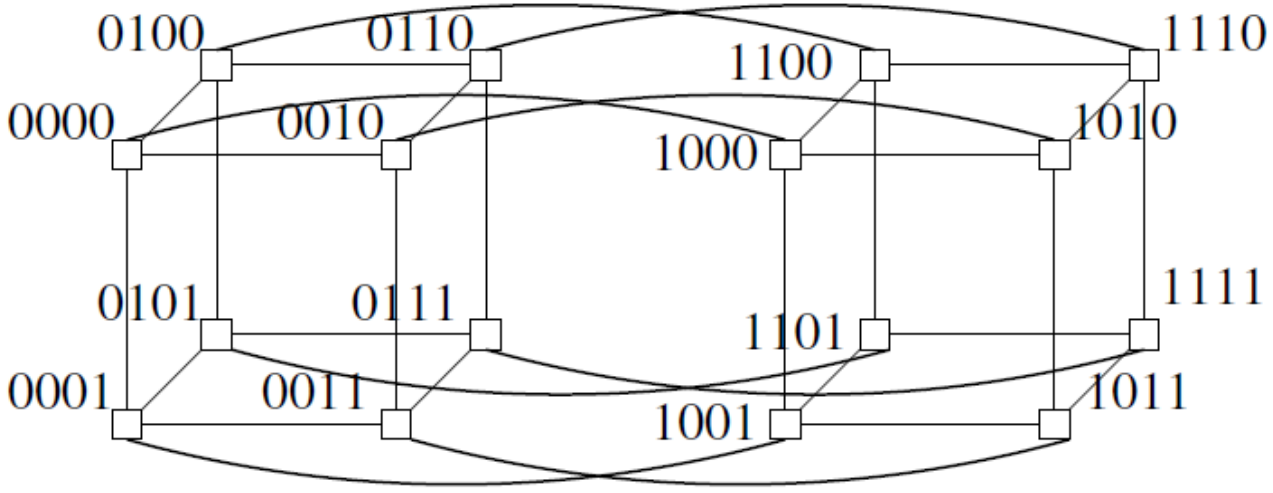
Multicomputer Parallel Systems



- k-dimensional hypercube has 2^k processor-and-memory units (nodes)
- each dimension is associated with a bit position in the label
- labels of two adjacent nodes differ in the k^{th} bit position for dimension k
- shortest path between any two processors → *Hamming distance*
- Hamming distance <= k

Multicomputer Parallel Systems

- routing in the hypercube is done hop-by-hop
- multiple routes exist between any pair of nodes → provides fault tolerance and congestion control mechanism



Parallelism/Speedup

- measure of the relative speedup of a specific program on a given machine
- depends on
  - number of processors
  - mapping of the code to the processors
- speedup = T(1)/T(n)
  - T(1) = time with a single processor
  - T(n) = time with n processors

Parallel/Distributed Program Concurrency

$$\frac{\text{number of local operations}}{\text{total number of operations}}$$

non-communication and non-shared memory access

includes the communication or shared memory access operations

Distributed Communication Models

- Remote Procedure Call (RPC)
- Publish/Subscribe Model

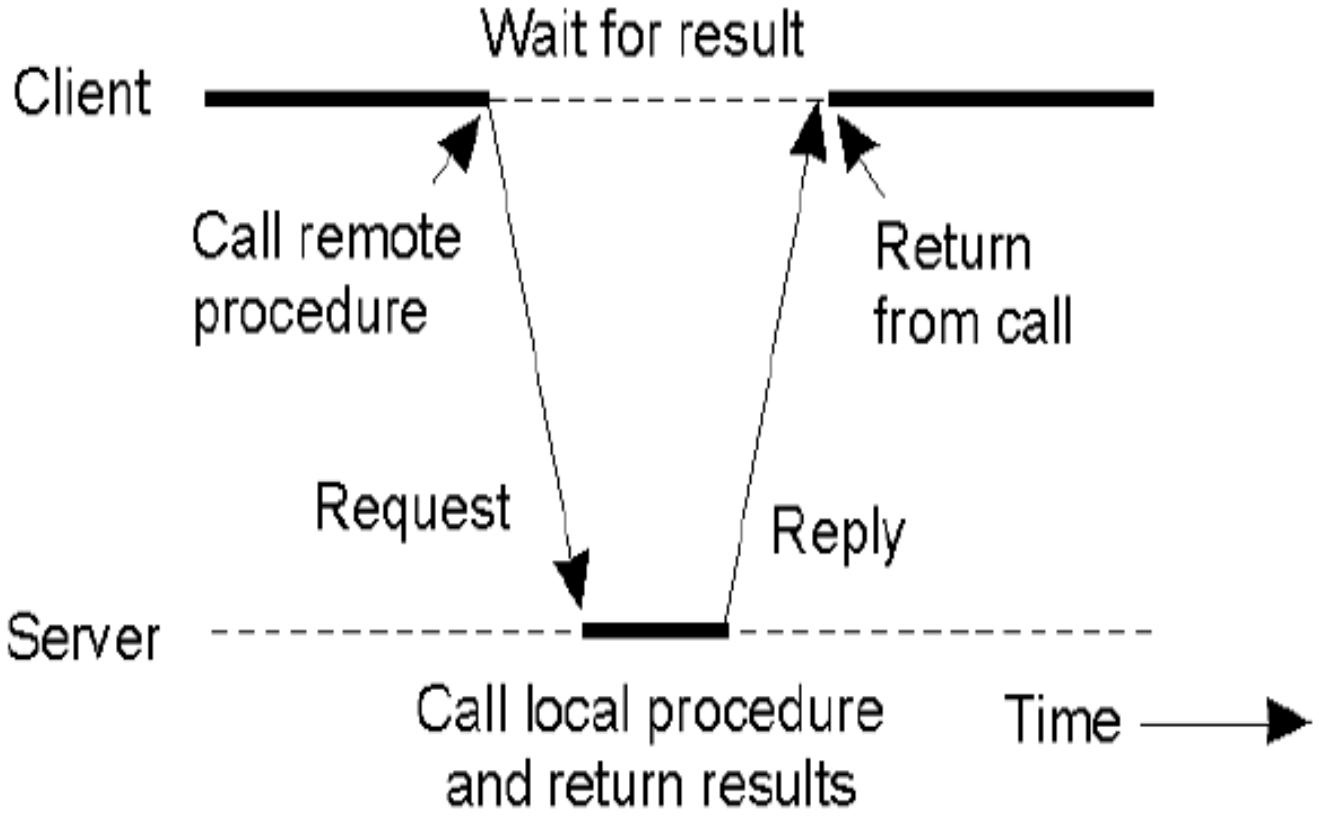
RPC

- Called procedure code may reside on a remote machine
- make remote procedure call appear like local procedure call
- goal is to hide the details of the network communication (namely, the sending and receiving of messages)
- calling procedure should not be aware that the called procedure is executing on a different machine

Source: http://web.cs.wpi.edu/~rek/DCS/D04/Communication.pdf

RPC between Client & Server

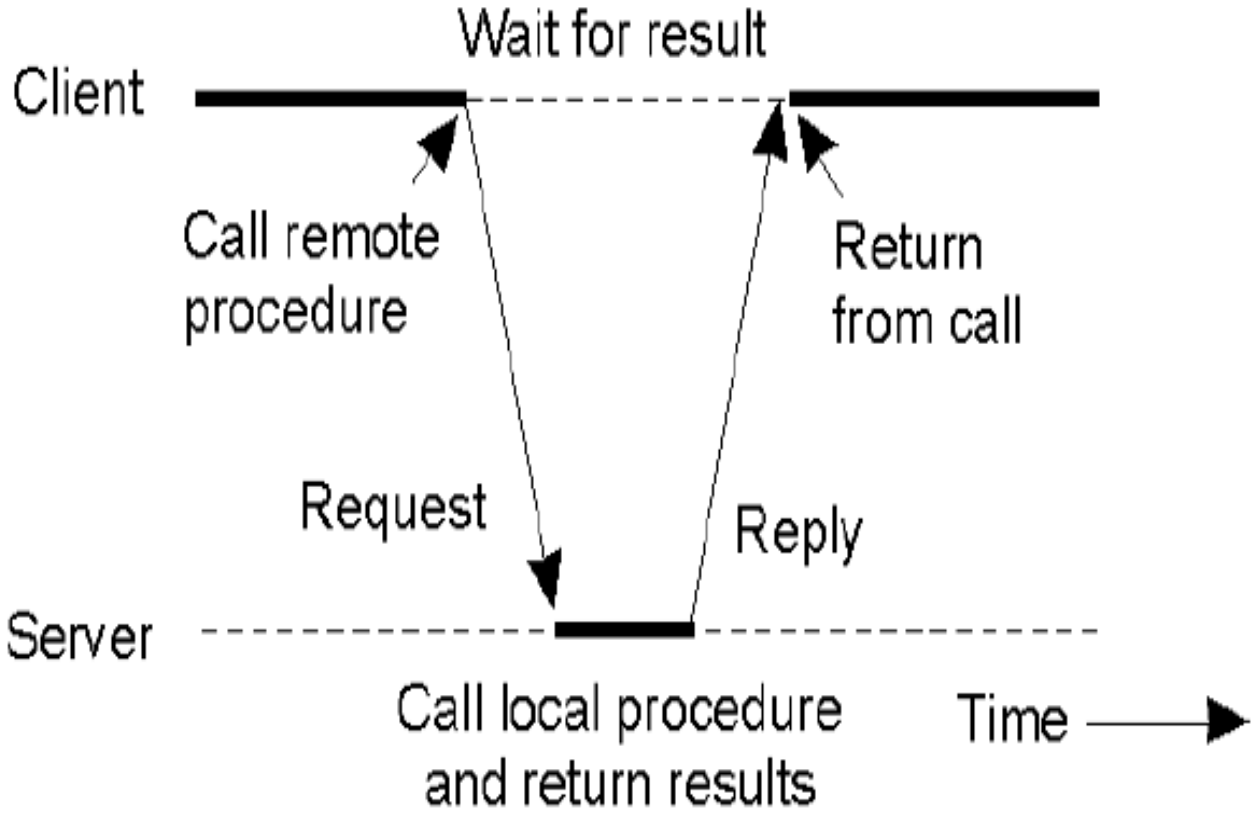
- The client procedure calls a client stub for passing parameters
- The client stub marshals the parameters, builds the message, and calls the local OS
- The client's OS sends the message to the remote OS
- The server's remote OS gives message to a server stub
- The server stub demarshals the parameters and calls the desired server routine



Source: http://web.cs.wpi.edu/~rek/DCS/D04/Communication.pdf

RPC between Client & Server

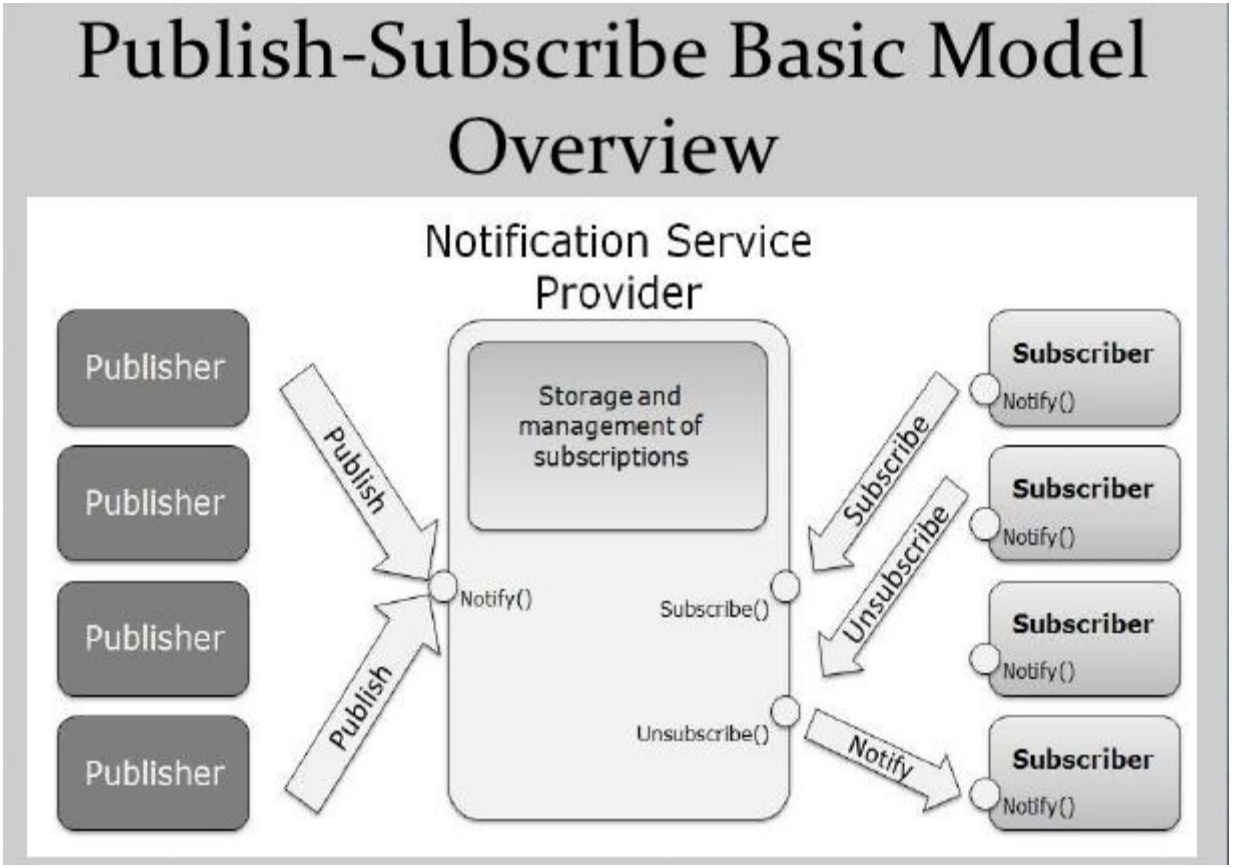
- The server routine does work and returns result to the server stub
- The server stub marshals the return values into the message and calls local OS
- The server OS sends the message to the client's OS
- The client's OS gives the message to the client stub
- The client stub demarshals the result, and execution returns to the client



Source: http://web.cs.wpi.edu/~rek/DCS/D04/Communication.pdf

Publish/ Subscribe Model

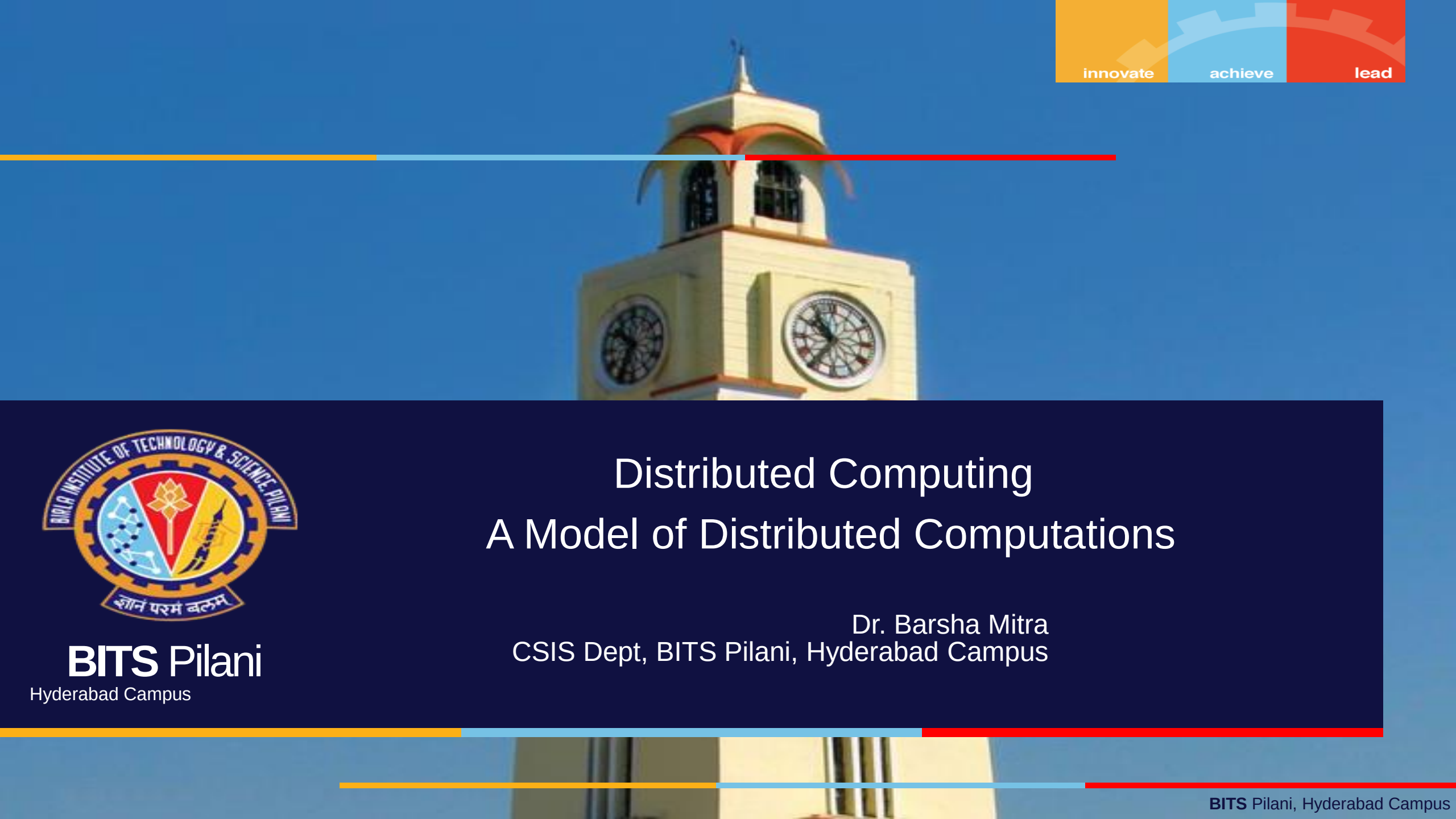
- allows any number of publishers to communicate with any number of subscribers asynchronously
- form of multicasting
- events are produced by publishers
- interested subscribers are notified when the events are available
- information is provided as notifications
- subscribers continue their tasks until they receive notifications



References

- Ajay D. Kshemkalyani, and Mukesh Singhal, Chapter 1, “Distributed Computing: Principles, Algorithms, and Systems”, Cambridge University Press, 2008.
- https://medium.com/@rashmishehana_48965/middleware-technologies-e5def95da4e
- http://wiki.c2.com/?PublishSubscribeModel
- https://www.slideshare.net/ishraqabd/publish-subscribe-model-overview-13368808/5

Thank You



innovate achieve lead

BITS Pilani
Hyderabad Campus

Distributed Computing
A Model of Distributed Computations

Dr. Barsha Mitra
CSIS Dept, BITS Pilani, Hyderabad Campus

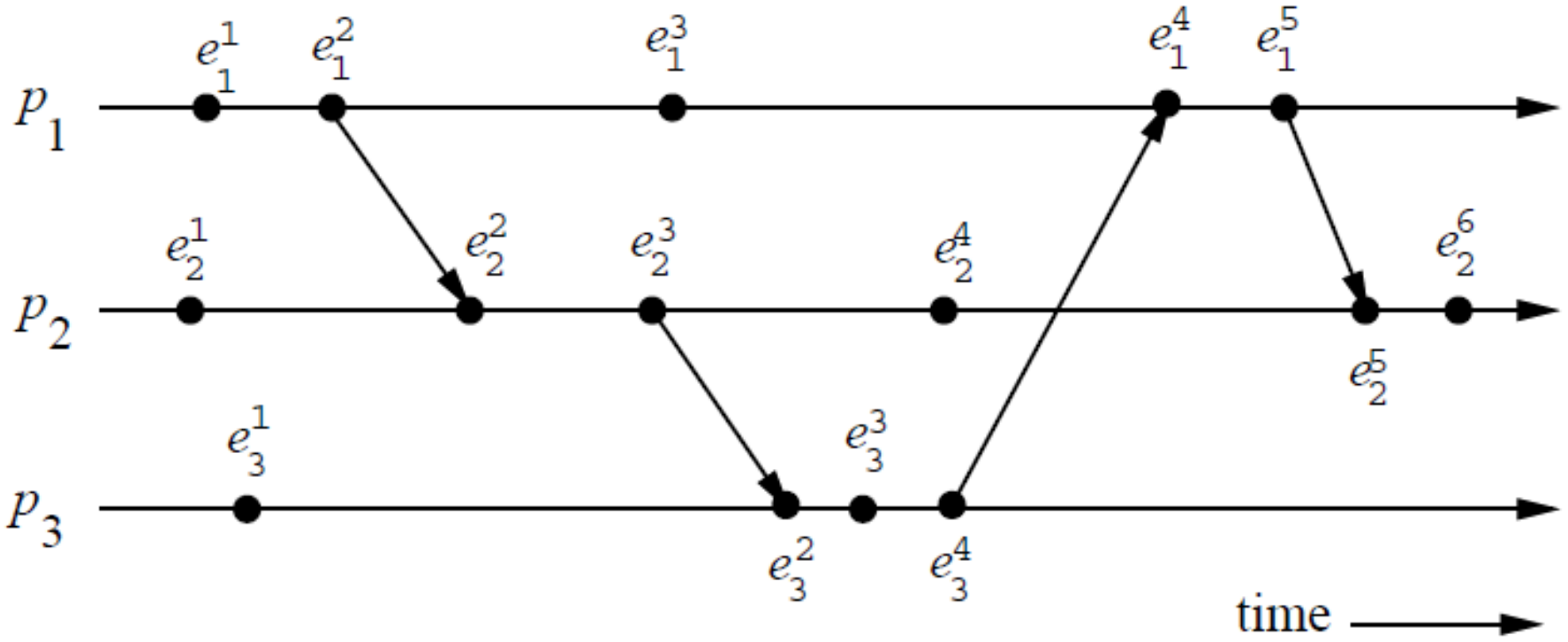
Preliminaries

- distributed system consists of a set of processors
- connected by a communication network
- communication delay is finite but unpredictable
- processors do not share a common global memory
- asynchronous message passing
- no common physical global clock
- communication medium may deliver messages out of order
- messages may be lost, garbled, or duplicated due to timeout and retransmission
- processors may fail, and communication links may go down

A Distributed Program

- **C_{ij}** : channel from process p_i to process p_j
- **m_{ij}**: message sent by p_i to p_j
- Global state of a distributed computation consists of
  - states of the processes
  - states of the communication channels
- State of a process is characterized by its local memory state
- Set of messages in transit in the channel characterizes state of channel
- Occurrence of events
  - changes respective processes' states
  - changes channels' states
  - causes transition in global system state

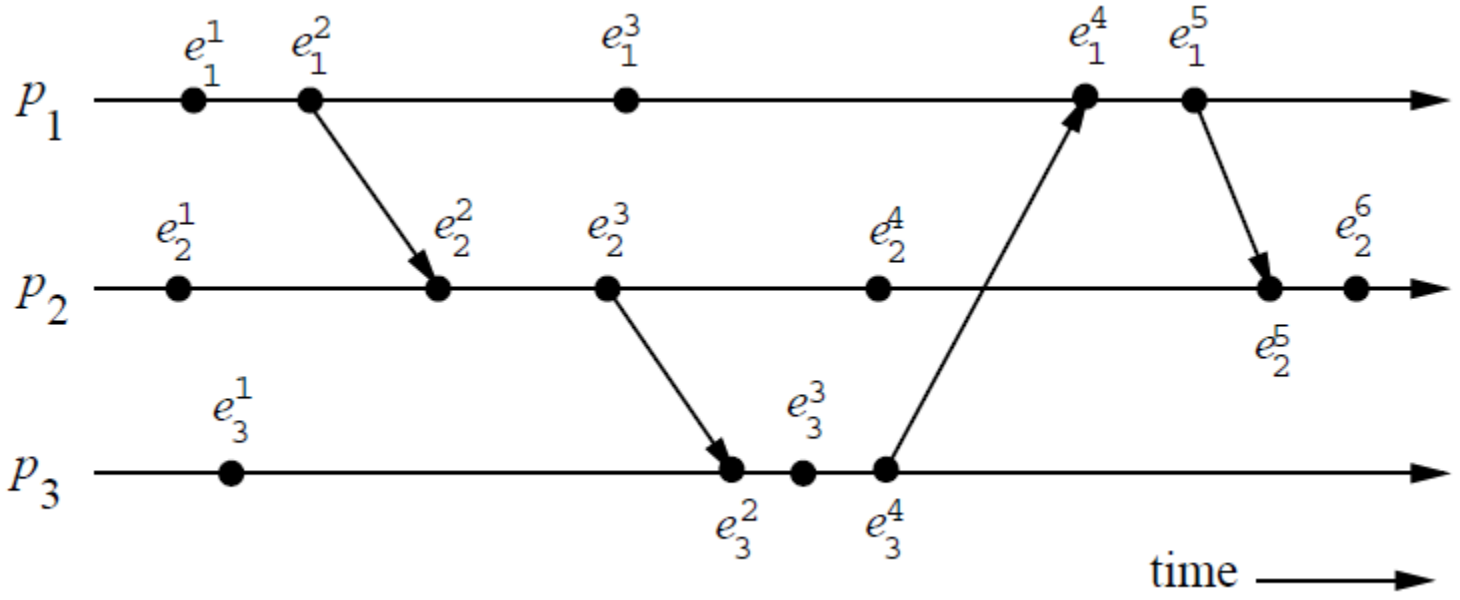
A Model of Distributed Executions



A Model of Distributed Executions

Causal Precedence/Dependency or Happens Before Relation

- Relation → denotes flow of information in a distributed computation
- $e_i \rightarrow e_j$ implies that all the information available at e_i is accessible at e_j
- path

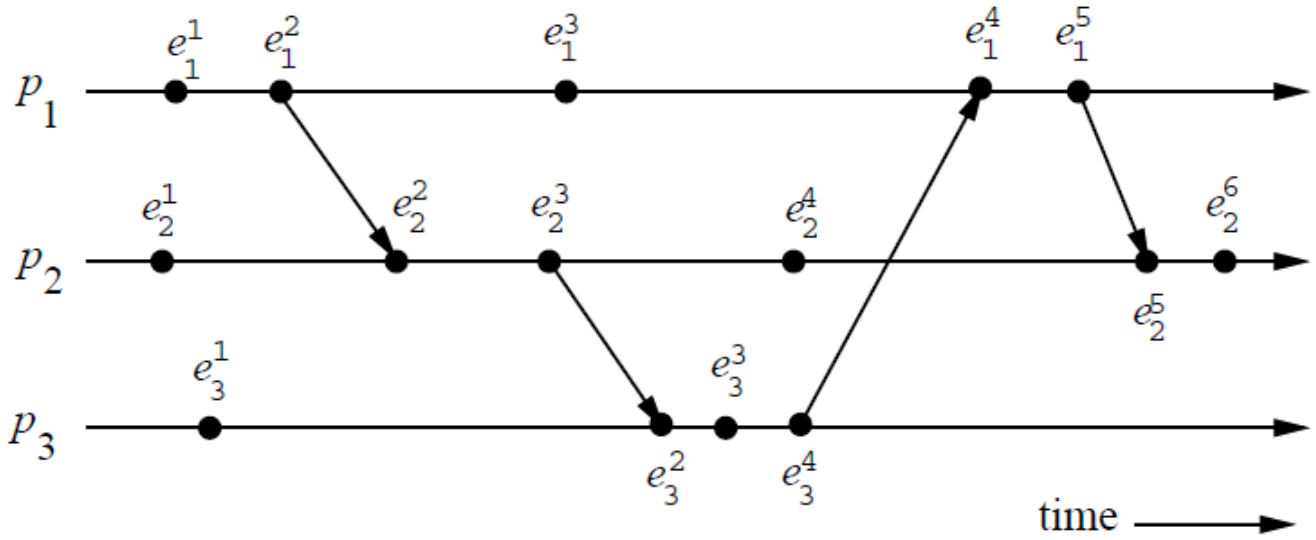


- $e_1^1 \rightarrow e_3^3$
- $e_3^3 \rightarrow e_2^6$

A Model of Distributed Executions

Concurrent Events

- For any two events e_i and e_j , if $e_i \not\rightarrow e_j$ and $e_j \not\rightarrow e_i$, then events e_i and e_j are concurrent (denoted as $e_i \,||\, e_j$)
- non-transitive



- $e_1^3 \,||\, e_3^3$
- $e_2^4 \,||\, e_3^1$

Models of Communication Networks

- FIFO model:
  - each channel acts as a first-in first-out message queue
  - message ordering is preserved by channel

- Non-FIFO model
  - channel acts like a set
  - sender process adds messages to channel
  - receiver process removes messages from it

Models of Communication Networks

- Causal ordering model is based on Lamport's "happens before" relation
- A system supporting causal ordering model satisfies :

  CO: for m_{ij} and m_{kj} , if send(m_{ij}) → send(m_{kj}),
  then rec(m_{ij}) → rec(m_{kj})
- Causally ordered delivery of messages implies FIFO message delivery
- CO ⊂ FIFO ⊂ Non-FIFO

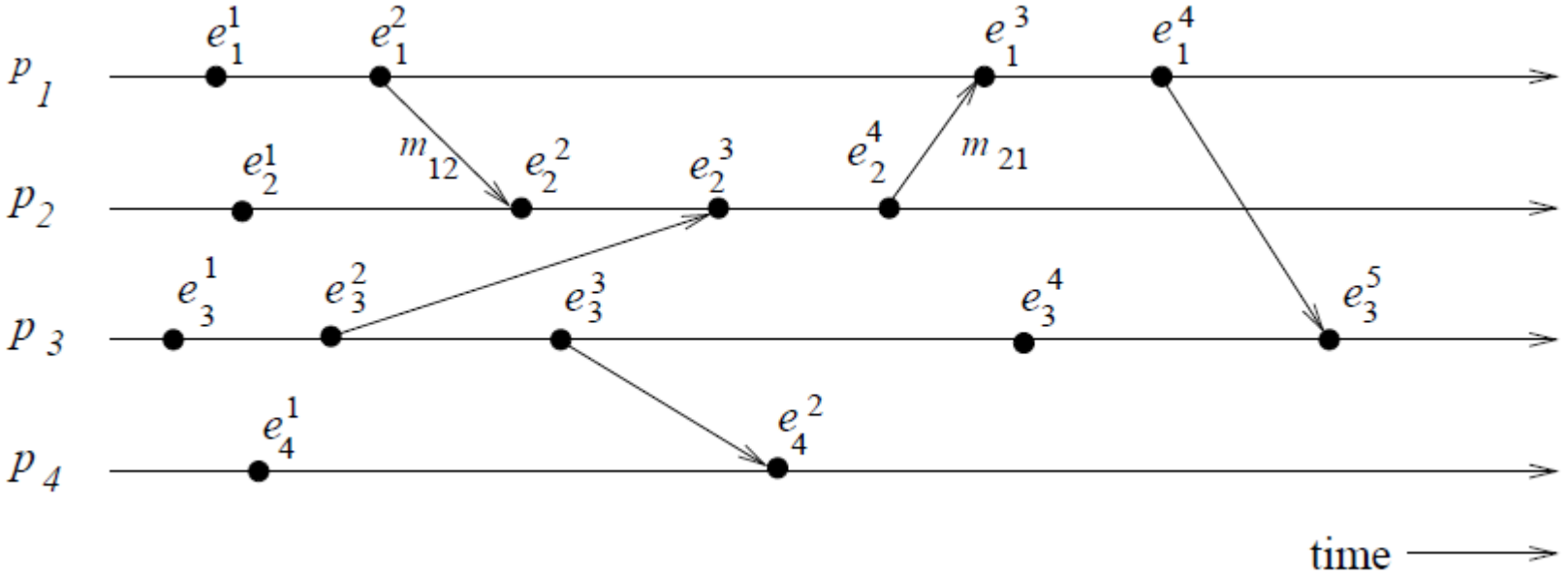
Global State of a Distributed System

- LS_i^x: state of process p_i after the occurrence of e_i^x and before e_i^{x+1}
- SC_{ij} : state of channel C_{ij}

Consistent Global State

- **a state will be meaningful and consistent provided every message that is recorded as received is also recorded as sent**
- **for all m_{ij}, $send(m_{ij}) \notin LS_i^x \Rightarrow m_{ij} \notin SC_{ij} \wedge rec(m_{ij}) \notin LS_j^y$**

Global State of a Distributed System



$GS_2 = \{LS_1^2, LS_2^4, LS_3^4, LS_4^2\}$

is consistent

Reason: all channels are empty except C_{21} which contains m_{21}

Cuts of a Distributed Computation

- Zigzag line joining one arbitrary point on each process line
- Set of events in distributed computation is partitioned into
  - PAST: contains all events to the left of the cut
  - FUTURE: contains all events to the right of the cut
- Every cut corresponds to a global state
- Every global state can be graphically represented as a cut

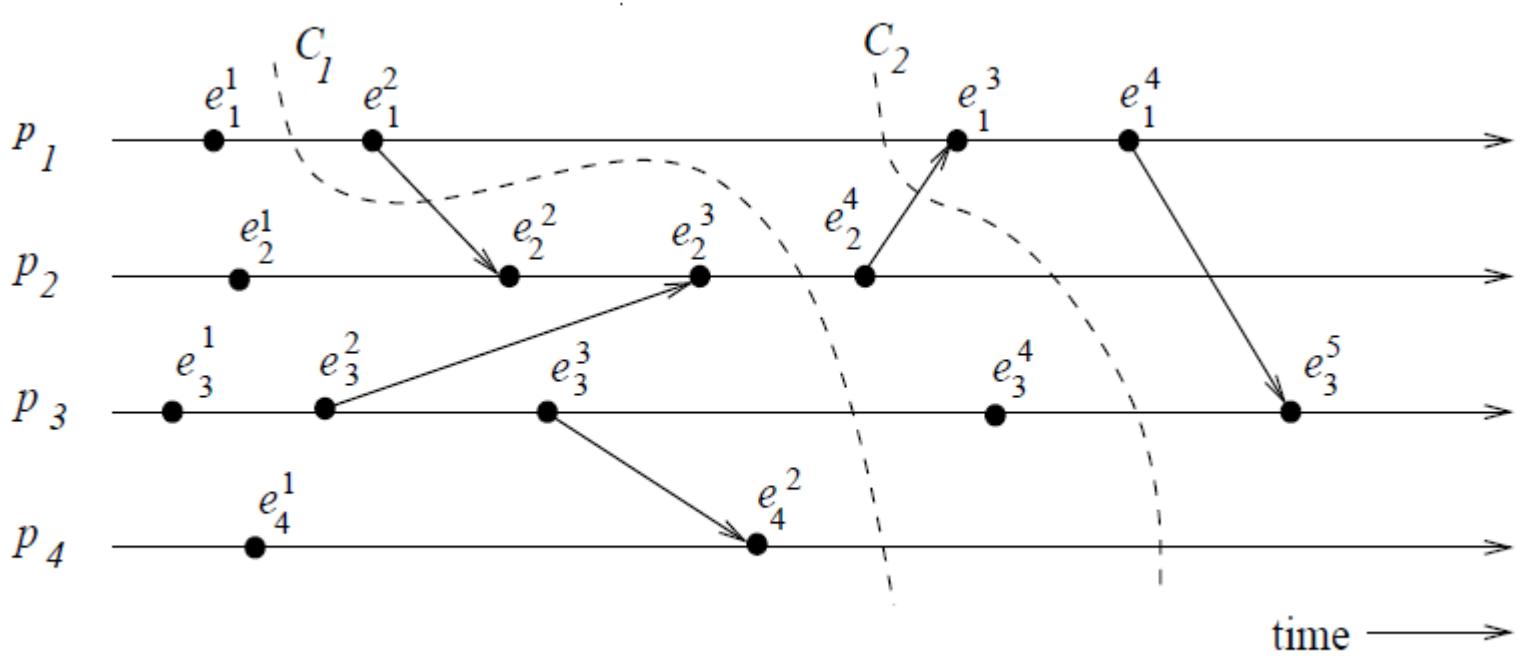
Cuts of a Distributed Computation

- **Consistent cut**: every message received in the PAST of the cut was sent in the PAST of that cut

- All messages that cross the cut from the PAST to the FUTURE are in transit

- **Inconsistent cut**: if a message crosses the cut from the FUTURE to the PAST

Cuts of a Distributed Computation

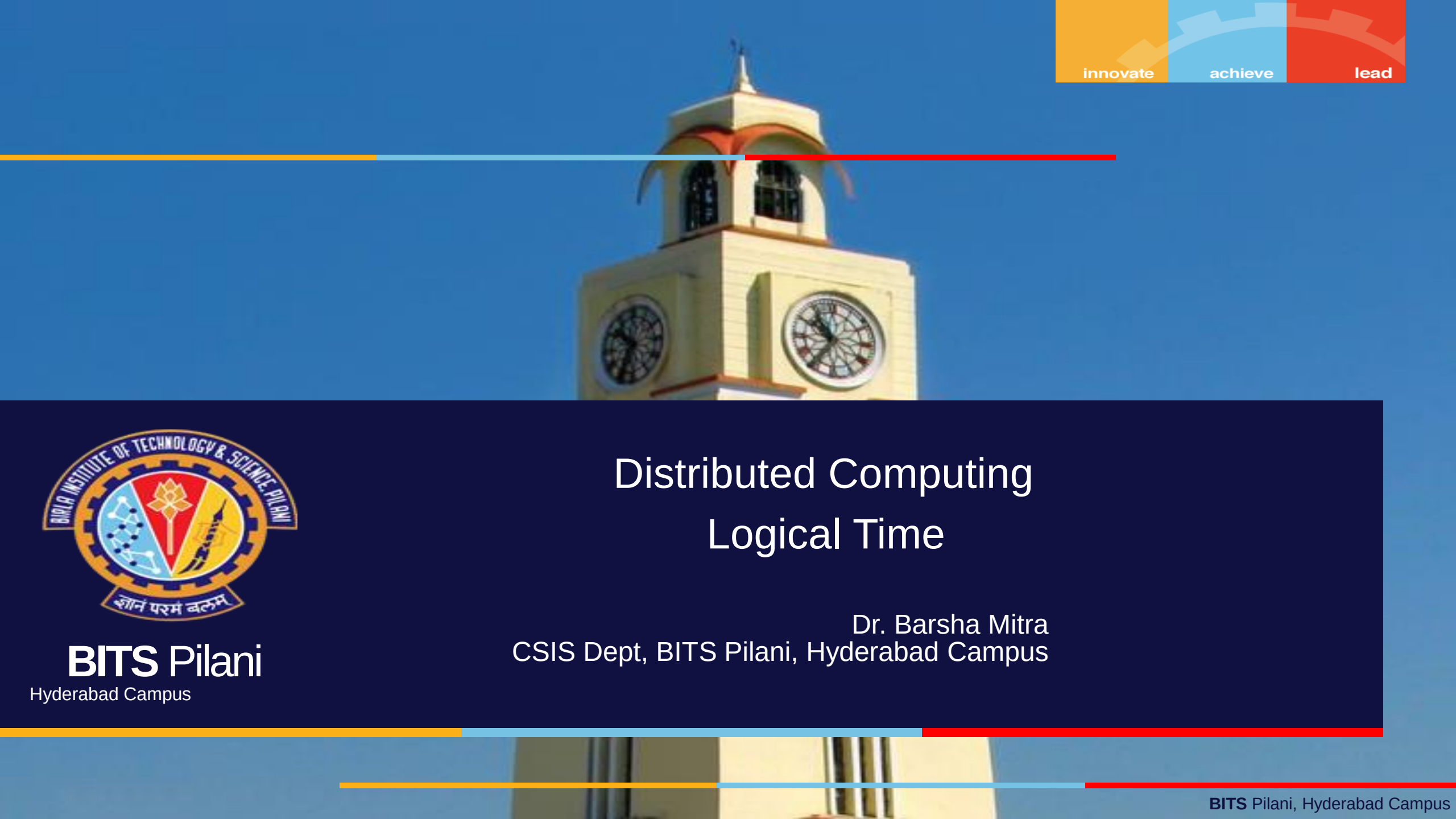


- C_2 → consistent cut
- C_1 → inconsistent cut

Reference

- Ajay D. Kshemkalyani, and Mukesh Singhal, Chapter 2, “Distributed Computing: Principles, Algorithms, and Systems”, Cambridge University Press, 2008.

Thank You



innovate achieve lead

BITS Pilani
Hyderabad Campus

Distributed Computing
Logical Time

Dr. Barsha Mitra
CSIS Dept, BITS Pilani, Hyderabad Campus

Introduction

- not possible to have global physical time
- realize only an approximation of physical time using logical time
- logical time advances in jumps
- captures monotonicity property

- monotonicity property: if event *a* causally affects event *b*, then timestamp of *a* is smaller than timestamp of *b*
- Ways to represent logical time:
  - scalar time
  - vector time

Logical Clocks

- every process has a logical clock
- logical clock is advanced using a set of rules
- each event is assigned a timestamp

Scalar Time - Definition

- time domain is the set of non-negative integers

- C_i:
  - integer variable
  - denotes logical clock of p_i

Scalar Time - Definition

Rules for updating clock:

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

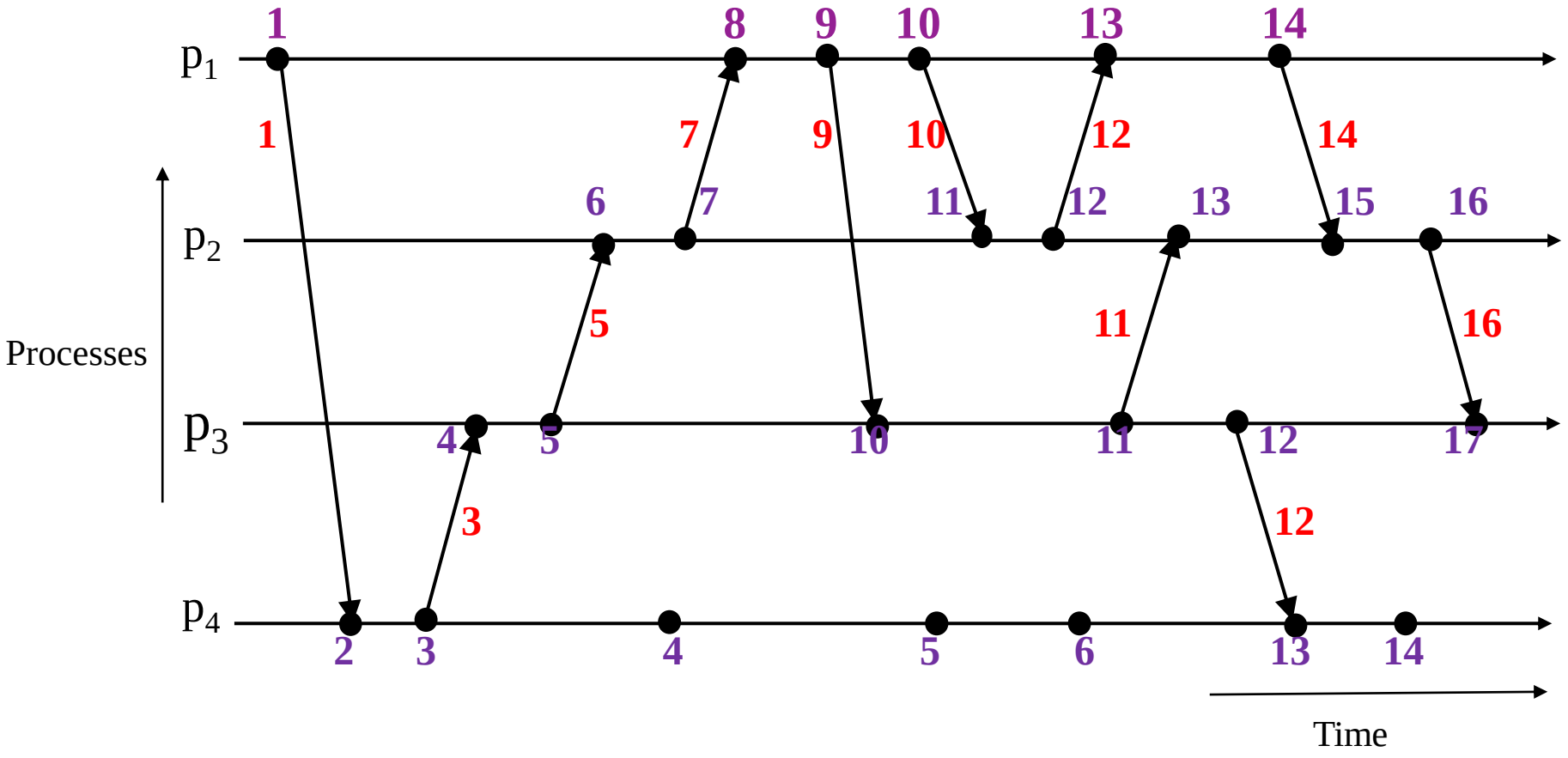
$$C_i := C_i + d \qquad (d > 0)$$

  - *d can have different values*
  - Usually $d = 1$

R2: When process p_i receives a message with timestamp C_{msg}, it executes the following actions:

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

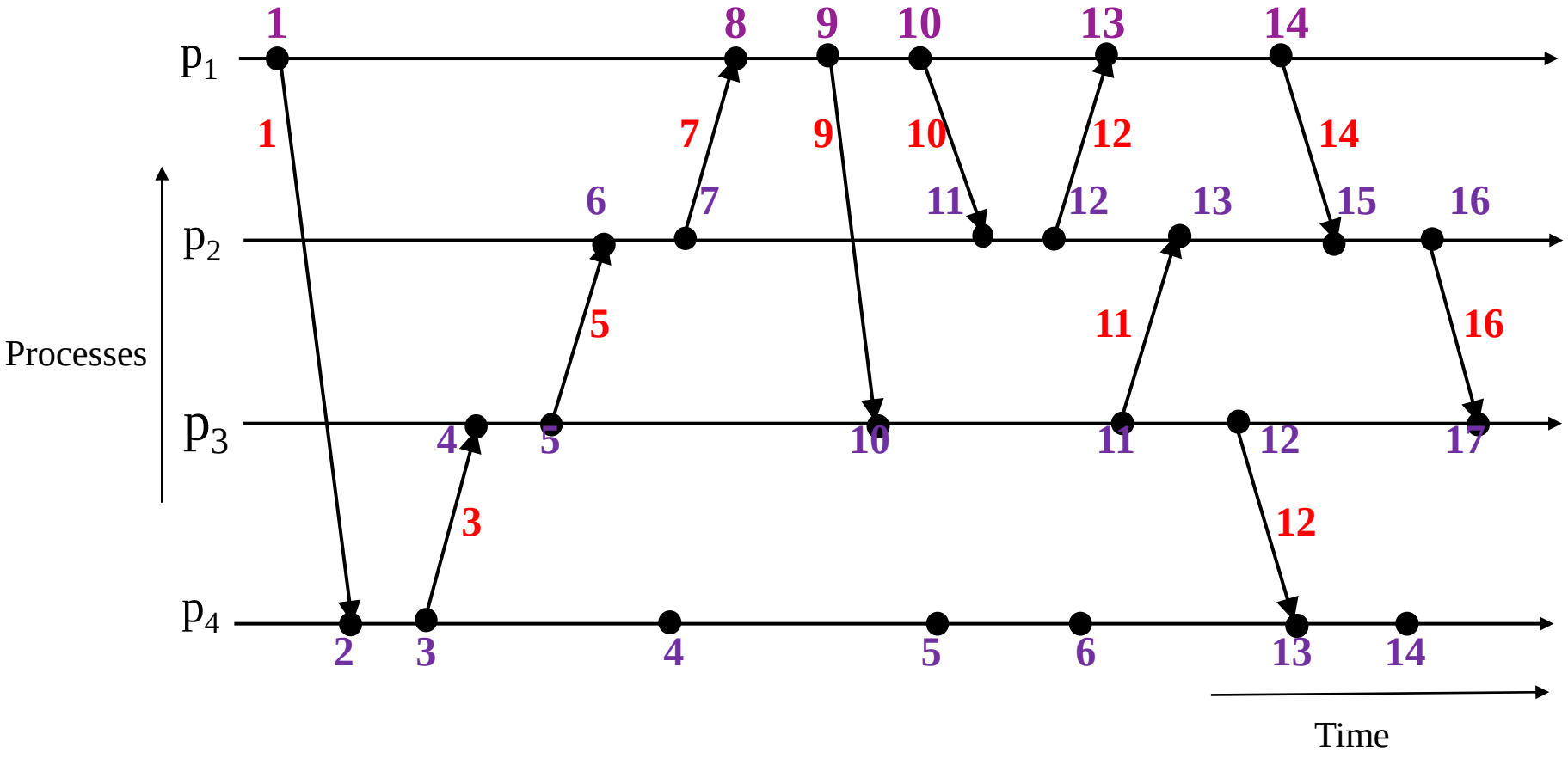
Evolution of Scalar Time



Scalar Time – Basic Properties

- Consistency
- Total Ordering
- Event Counting
- No Strong Consistency
- From Recorded Lecture

Scalar Time – Basic Properties

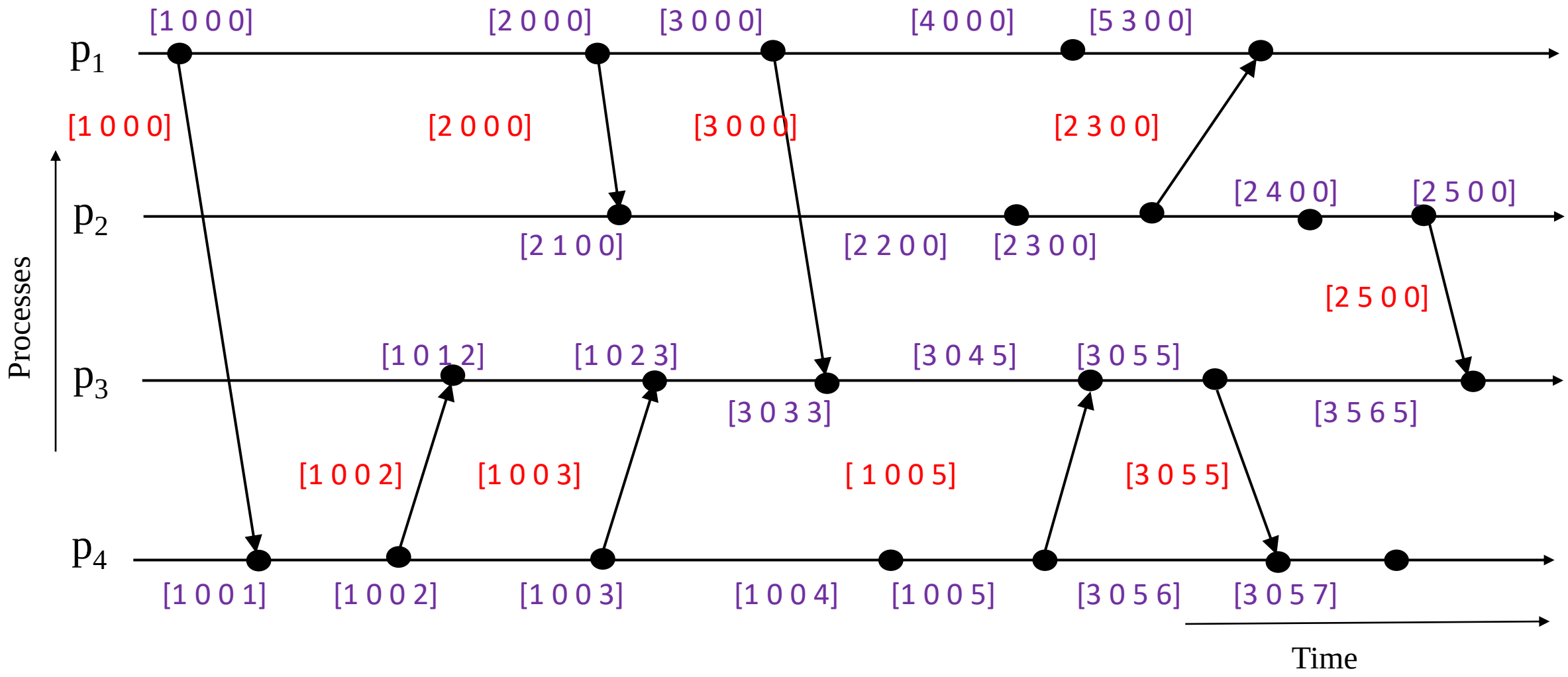


- d = 1
- height of event

Vector Time - Definition

- time domain is represented by a set of *n*-dimensional non-negative integer vectors
- $vt_i[i]$, $vt_i[j]$

Vector Time - Definition



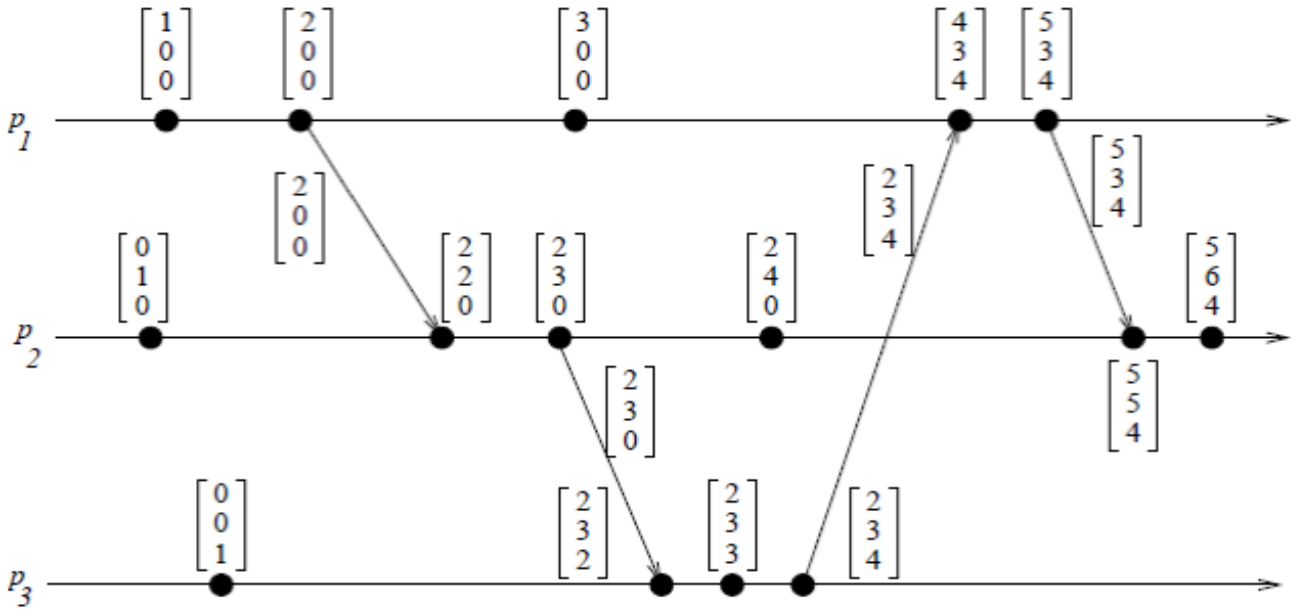
Vector Time

- Comparing Vector timestamps from Recorded Lecture
- Vector Time properties from Recorded Lecture

Vector Time – Basic Properties

Event counting

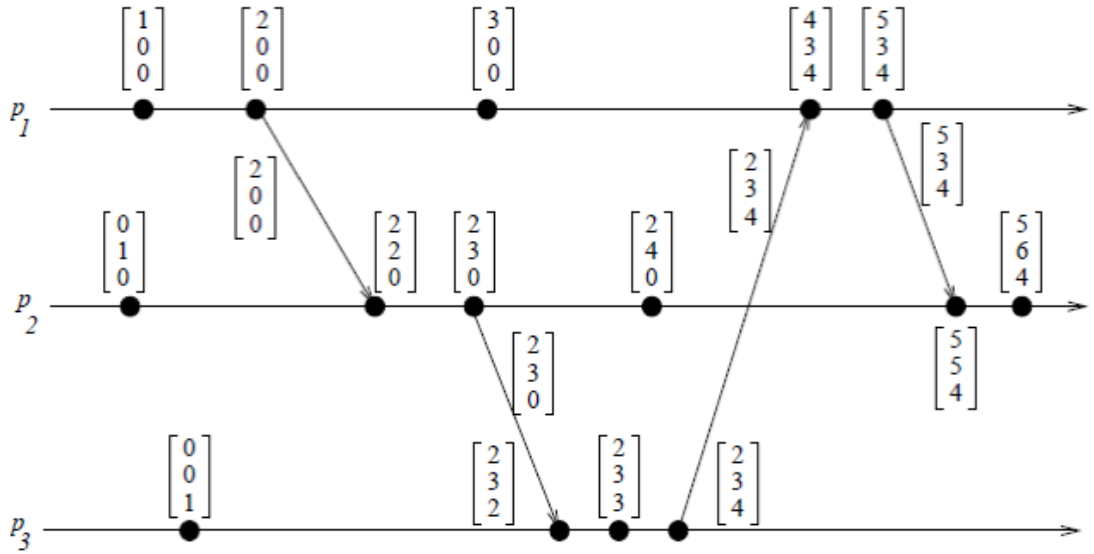
- if d is always 1 and $vt_i[i]$ is the i^{th} component of vector clock at process p_i
  - then $vt_i[i]$ = no. of events that have occurred at p_i until that instant



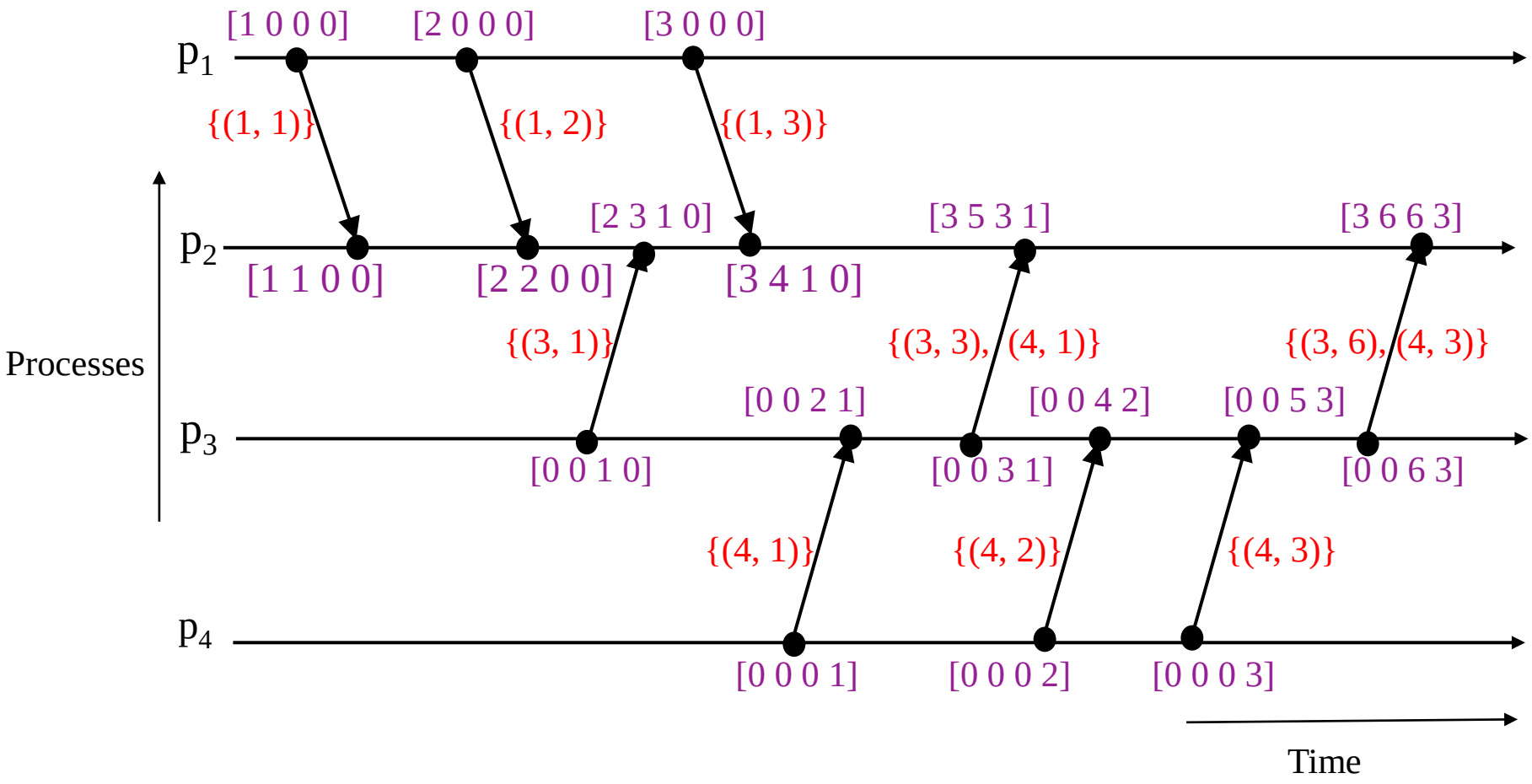
Vector Time – Basic Properties

Event counting

- if an event e has timestamp vh
  - $vh[j]$ = number of events executed by process p_j that causally precede e
  - $\Sigma\, vh[j] - 1$: total no. of events that causally precede e in the distributed computation



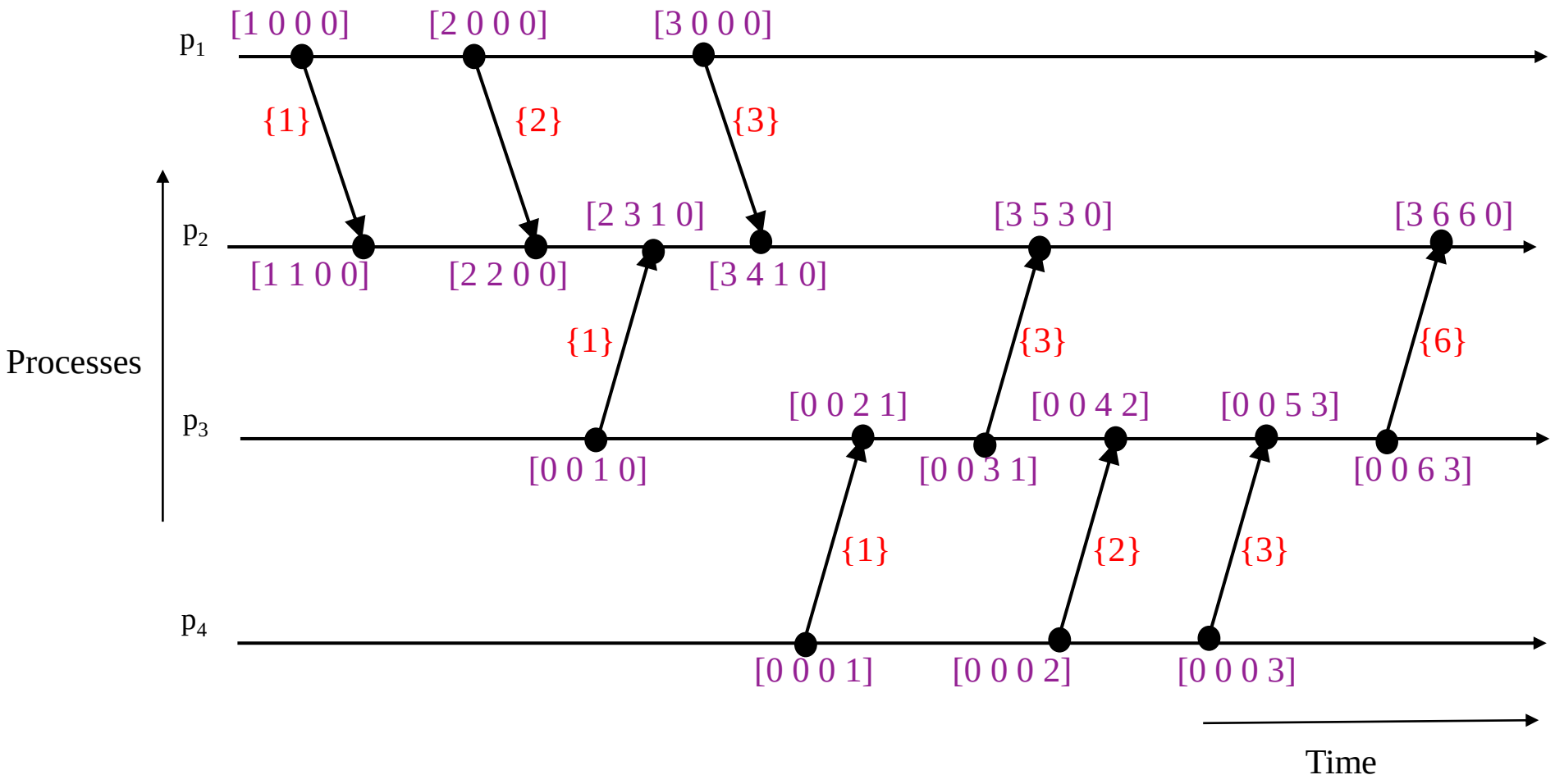
Singhal–Kshemkalyani's Differential Technique



Fowler–Zwaenepoel's Direct-Dependency Technique

- reduces the size of messages by transmitting only a scalar value
- process only maintains information regarding direct dependencies on other processes
- transitive (indirect) dependencies are not maintained by this method

Fowler–Zwaenepoel's Direct-Dependency Technique



Reference

- Ajay D. Kshemkalyani, and Mukesh Singhal, Chapter 3, “Distributed Computing: Principles, Algorithms, and Systems”, Cambridge University Press, 2008.

Thank You

innovate achieve lead

BITS Pilani
Hyderabad Campus

Distributed Computing
Global State & Snapshot Recording Algorithms

Dr. Barsha Mitra
CSIS Dept, BITS Pilani, Hyderabad Campus

Introduction

- problems in recording global state
  - lack of a globally shared memory
  - lack of a global clock
  - message transmission is asynchronous
  - message transfer delays are finite but unpredictable

System Model

- if a snapshot recording algorithm records the states of p_i and p_j as LS_i and LS_j , respectively, it must record the state of channel C_{ij} as transit(LS_i, LS_j)
- For C_{ij} , intransit messages are:

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

A Consistent Global State

- global state GS is a ***consistent global state*** iff:
  - **C1:** every message m_{ij} that is recorded as sent in the local state of sender p_i must be captured either in the state of C_{ij} or in the collected local state of the receiver p_j
  - **C1 states the law of conservation of messages:**
  - **C2:** $send(m_{ij}) \notin LS_i \Rightarrow m_{ij} \notin SC_{ij} \wedge rec(m_{ij}) \notin LS_j$
  - **Condition C2 states that for every effect, its cause must be present**

Chandy–Lamport Algorithm

Marker sending rule for process p_i

(1) Process p_i records its state.

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

Marker receiving rule for process p_j

On receiving a marker along channel C:

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

Record the state of C as the empty set

Execute the "marker sending rule"

else

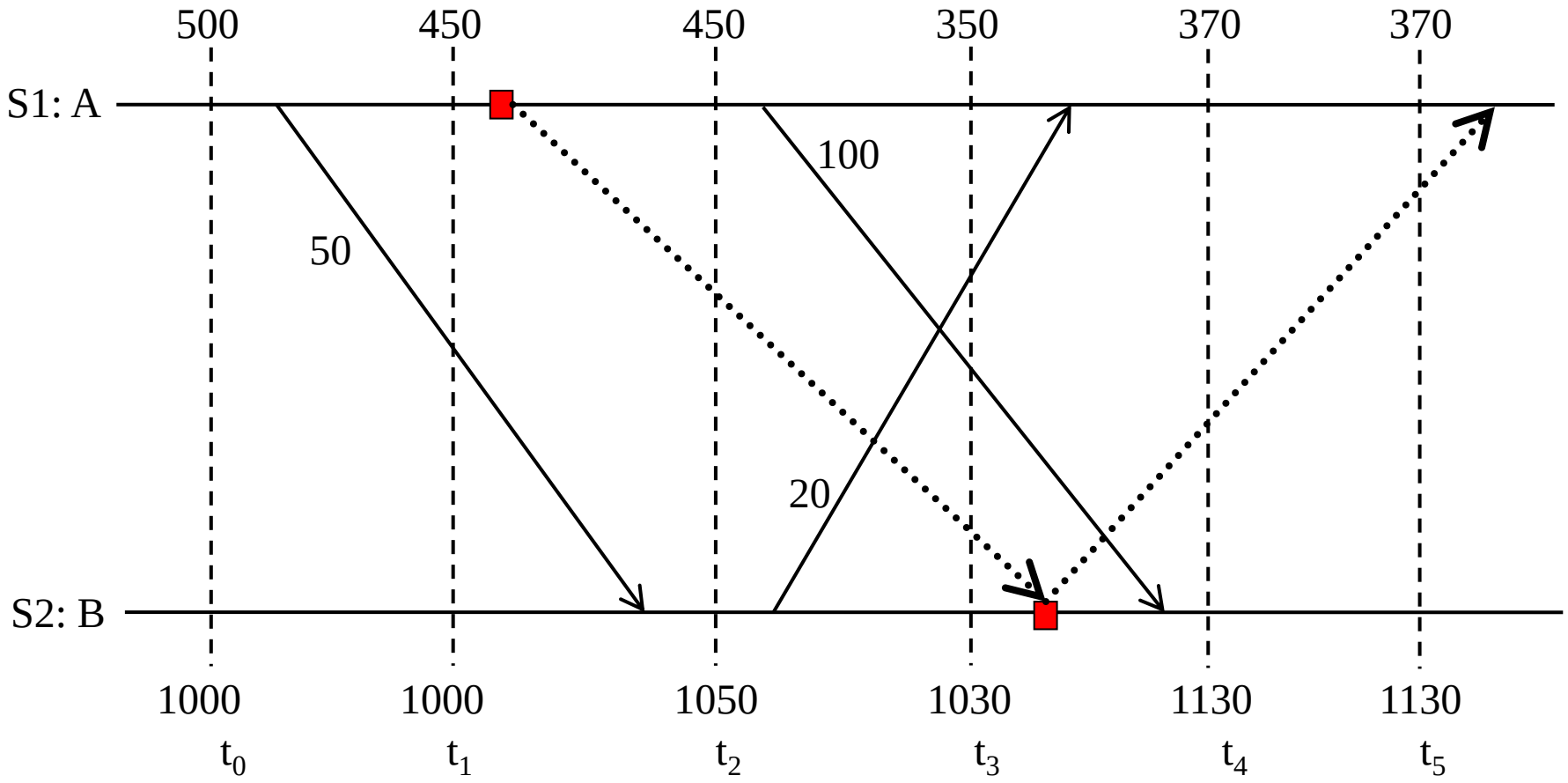
Record the state of C as the set of messages received along C after $p_{j's}$ state was recorded and before p_j received the marker along C

Chandy–Lamport Algorithm

Putting together recorded snapshots - each process can send its local snapshot to the initiator of the algorithm

termination criterion - each process has received a marker on all of its incoming channels

Chandy–Lamport Algorithm



Snapshot Algorithms for Non-FIFO Channels

- a marker cannot be used to
- non-FIFO algorithm by Lai and Yang use message piggybacking to distinguish computation messages sent after the marker from those sent before the marker

Lai–Yang Algorithm

- **coloring scheme**
- every process is initially white
- process turns red while taking a snapshot
- equivalent of the “marker sending rule” is executed when a process turns red
- every message sent by a white process is colored white
  - a white message is a message that was sent before the sender of that message recorded its local snapshot

Lai–Yang Algorithm

- every message sent by a red process is colored red
  - a red message is a message that was sent after the sender of that message recorded its local snapshot

- every white process takes its snapshot no later than the instant it receives a red message
- when a white process receives a red message, it records its local snapshot before processing the message

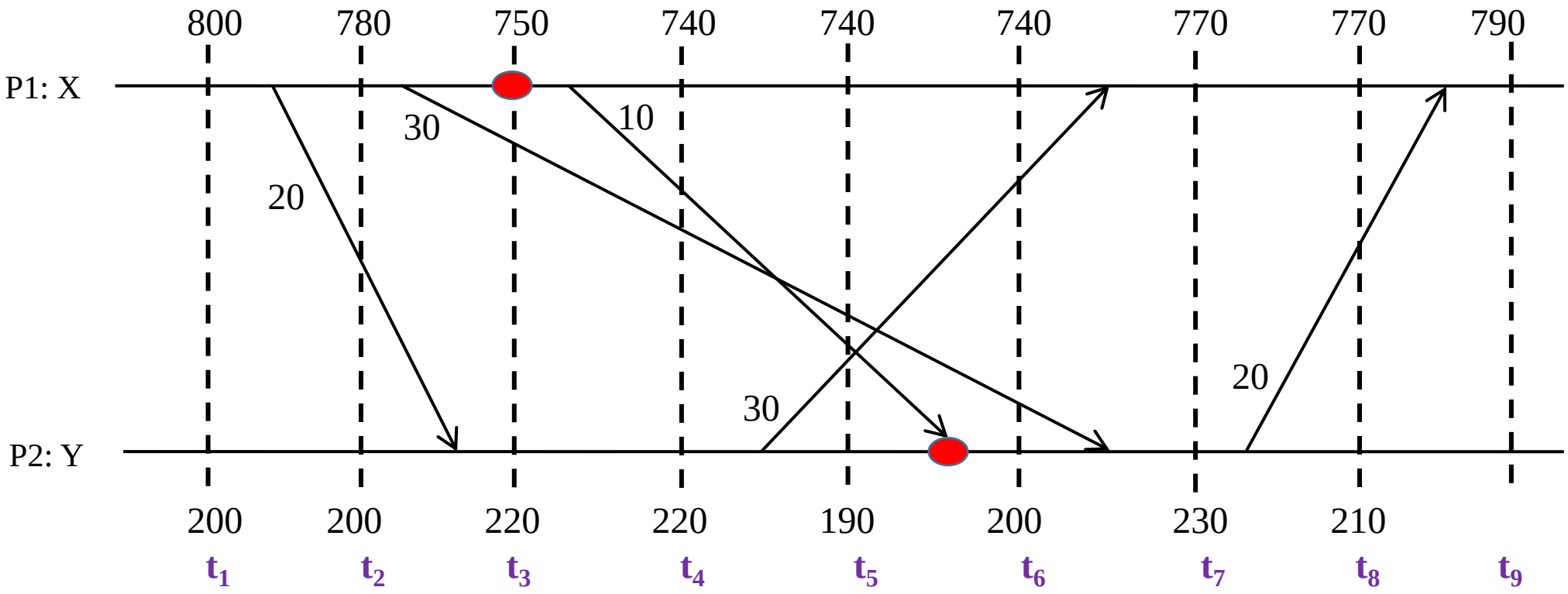
Lai–Yang Algorithm

- every white process records a history of all white messages sent or received by it along each channel
- when a process turns red, it sends these histories along with its snapshot to the initiator process that collects the global snapshot
- initiator process evaluates transit(LS_i, LS_j) to compute the state of a channel C_{ij} as:

SC_{ij} = {white messages sent by p_i on C_{ij}} – {white messages received by p_j on C_{ij}}

= $\{m_{ij} \mid send(m_{ij}) \in LS_i\} - \{m_{ij} \mid rec(m_{ij}) \in LS_j\}$

Lai–Yang Algorithm



Reference

- Ajay D. Kshemkalyani, and Mukesh Singhal, Chapter 4, “Distributed Computing: Principles, Algorithms, and Systems”, Cambridge University Press, 2008.

Thank You

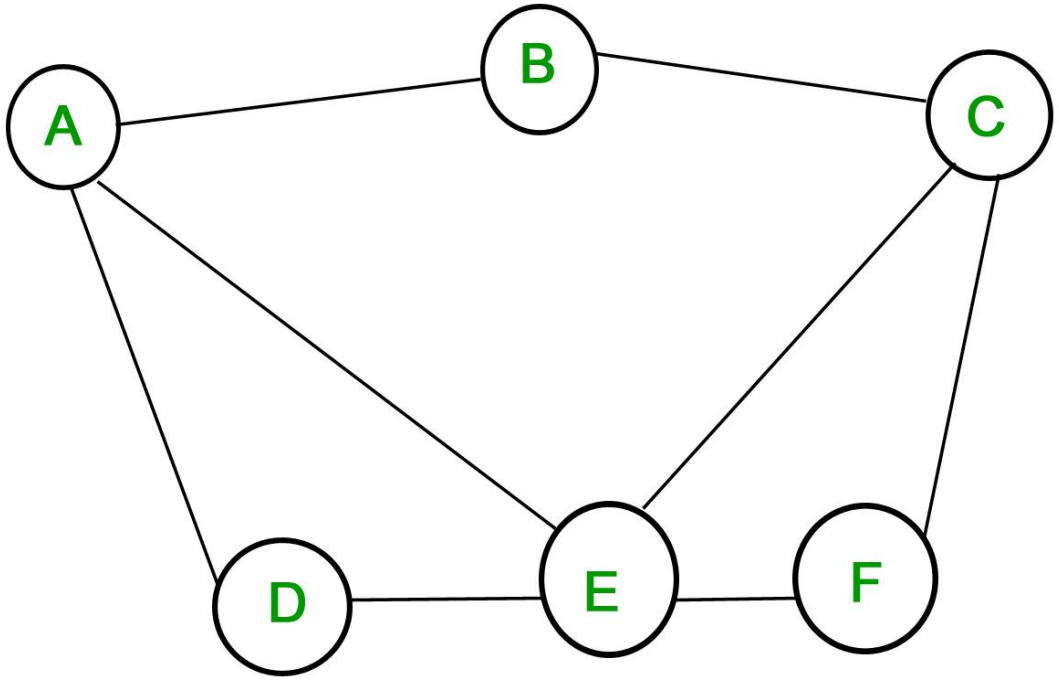
innovate achieve lead

Distributed Computing
Terminology & Basic Algorithms

Dr. Barsha Mitra
CSIS Dept, BITS Pilani, Hyderabad Campus

BITS Pilani
Hyderabad Campus

Notation & Definitions



- undirected unweighted graph G = (*N, L*) represents topology
- vertices are nodes
- edges are channels
- $n = |N|$
- $l = |L|$
- diameter of a graph –
  - minimum number of edges that need to be traversed to go from any node to any other node
  - diameter = $max_{i, j \in N}$ {length of the shortest path between *i* and *j*}

Synchronous Single-Initiator Spanning Tree Algorithm using Flooding

- algorithm proceeds in rounds (synchronous)
- root initiates the algorithm
- flooding of QUERY messages
- produce a spanning tree rooted at the root node
- each process P_i ($P_i \neq$ root) should output its own parent for the spanning tree

Synchronous Single-Initiator Spanning Tree Algorithm using Flooding

Local Variables maintained at each P_i

- **int** *visited*
- **int** *depth*
- **int** *parent*
- **set of int** *Neighbors*

Initially at each P_i

- *visited* = 0
- *depth* = 0
- *parent* = NULL
- *Neighbors* = set of neighbors

Synchronous Single-Initiator Spanning Tree Algorithm using Flooding

Algorithm for P_i

Round r = 1

if P_i = root **then**

visited = 1

depth = 0

send QUERY to *Neighbors*

if P_i receives a QUERY message **then**

visited = 1

depth = r

parent = root

plan to send QUERYs to *Neighbors* at next round

Synchronous Single-Initiator Spanning Tree Algorithm using Flooding

Algorithm for P_i

Round r > 1 and r <= *diameter*

if P_i planned to send in previous round **then**

P_i sends QUERY to *Neighbors*

if *visited* = 0 **then**

if P_i receives QUERY messages **then**

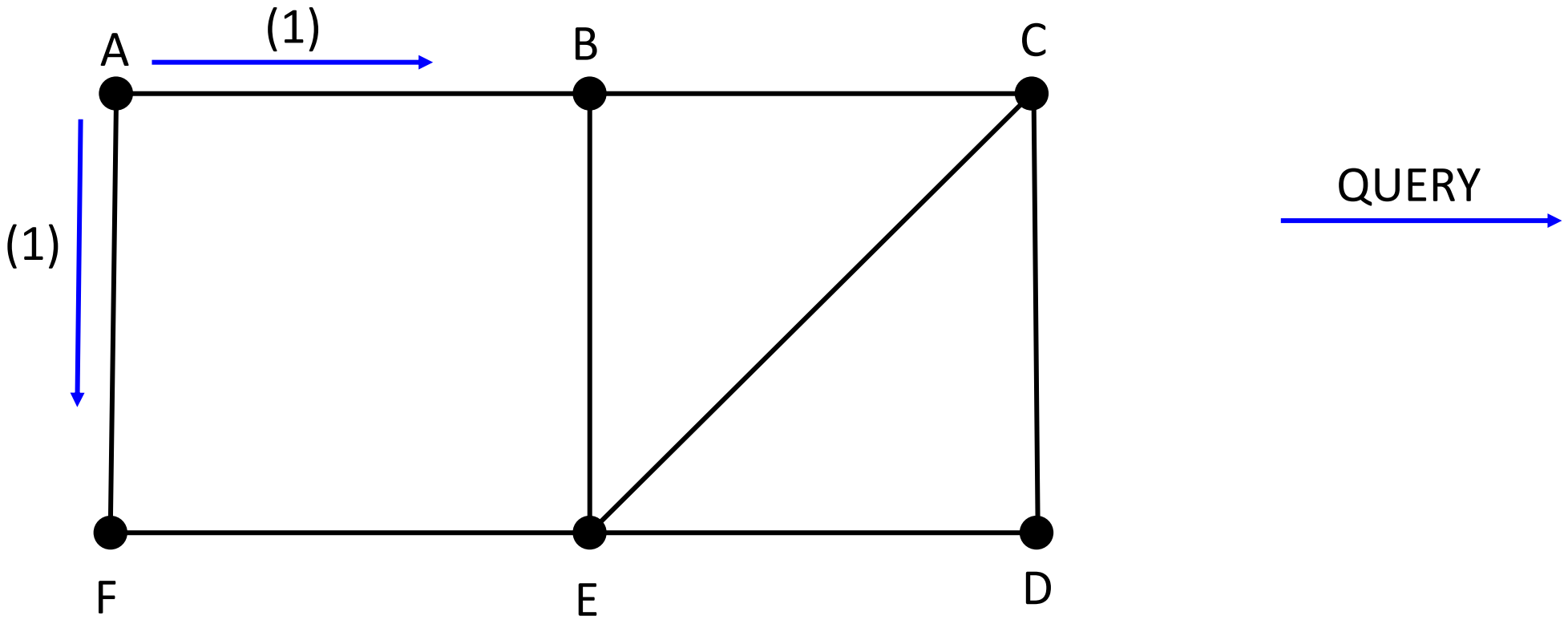
visited = 1

depth = r

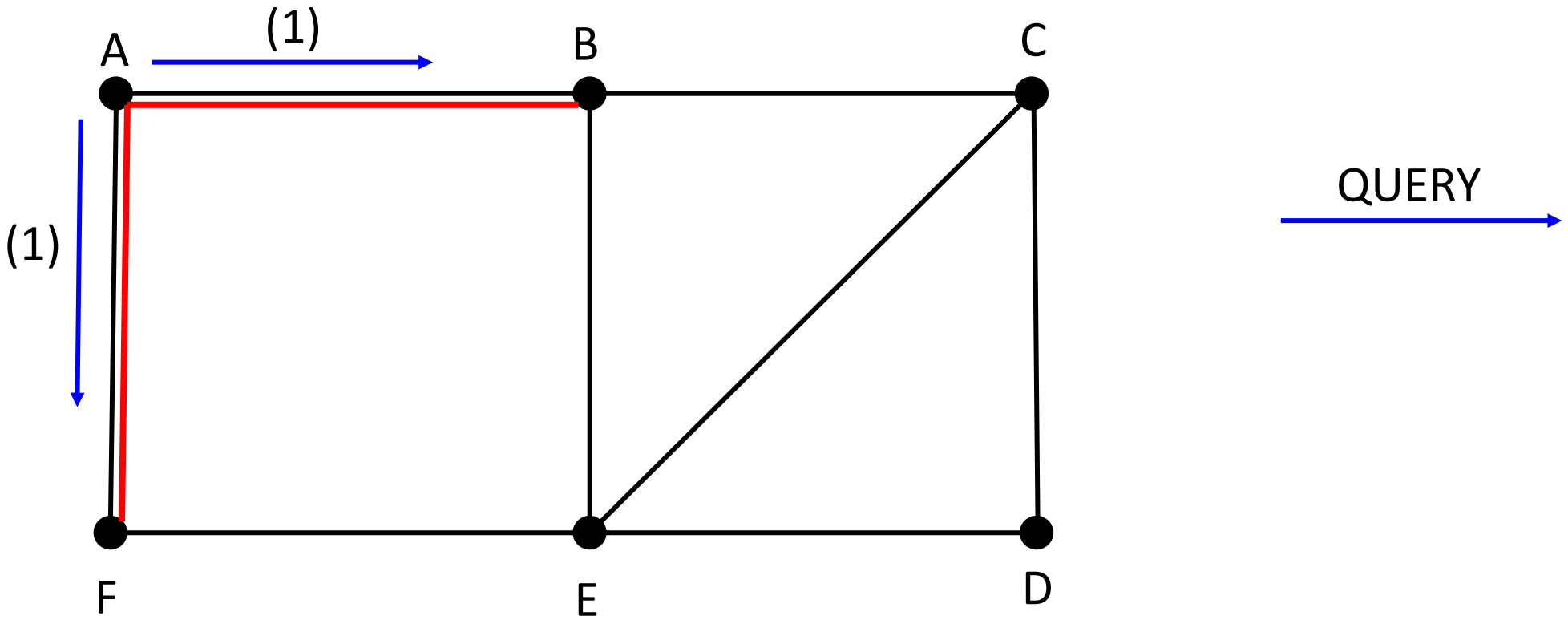
parent = any randomly selected node from which QUERY was received

plan to send QUERY to *Neighbors* \ {senders of QUERYs received in r}

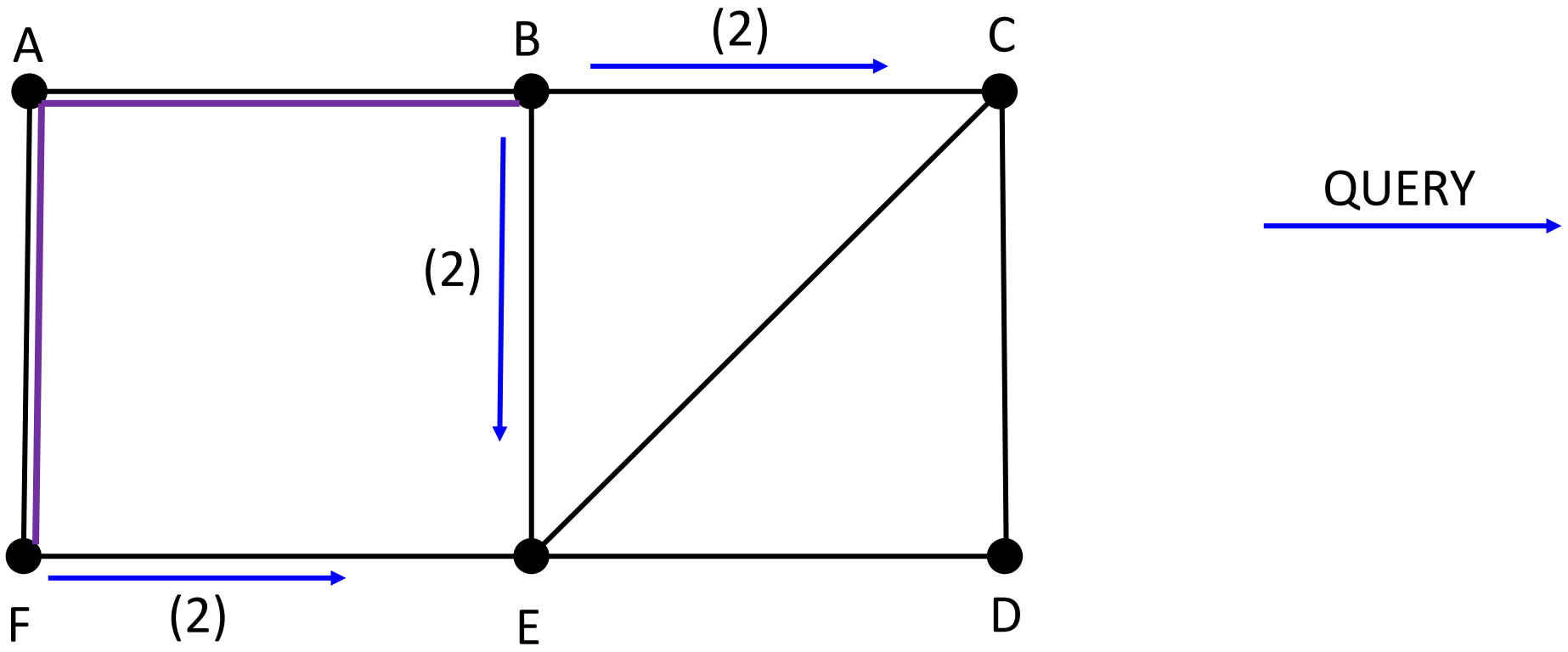
Synchronous Single-Initiator Spanning Tree Algorithm using Flooding



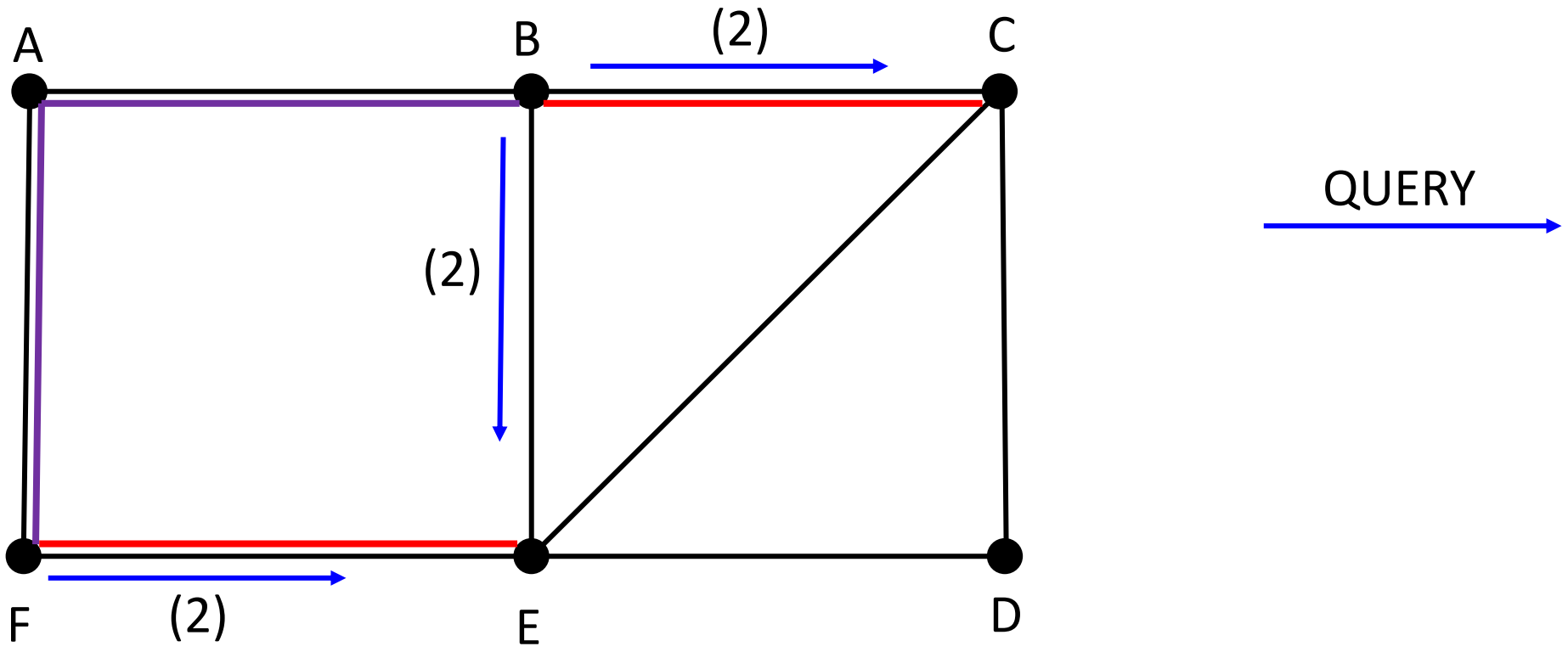
Synchronous Single-Initiator Spanning Tree Algorithm using Flooding



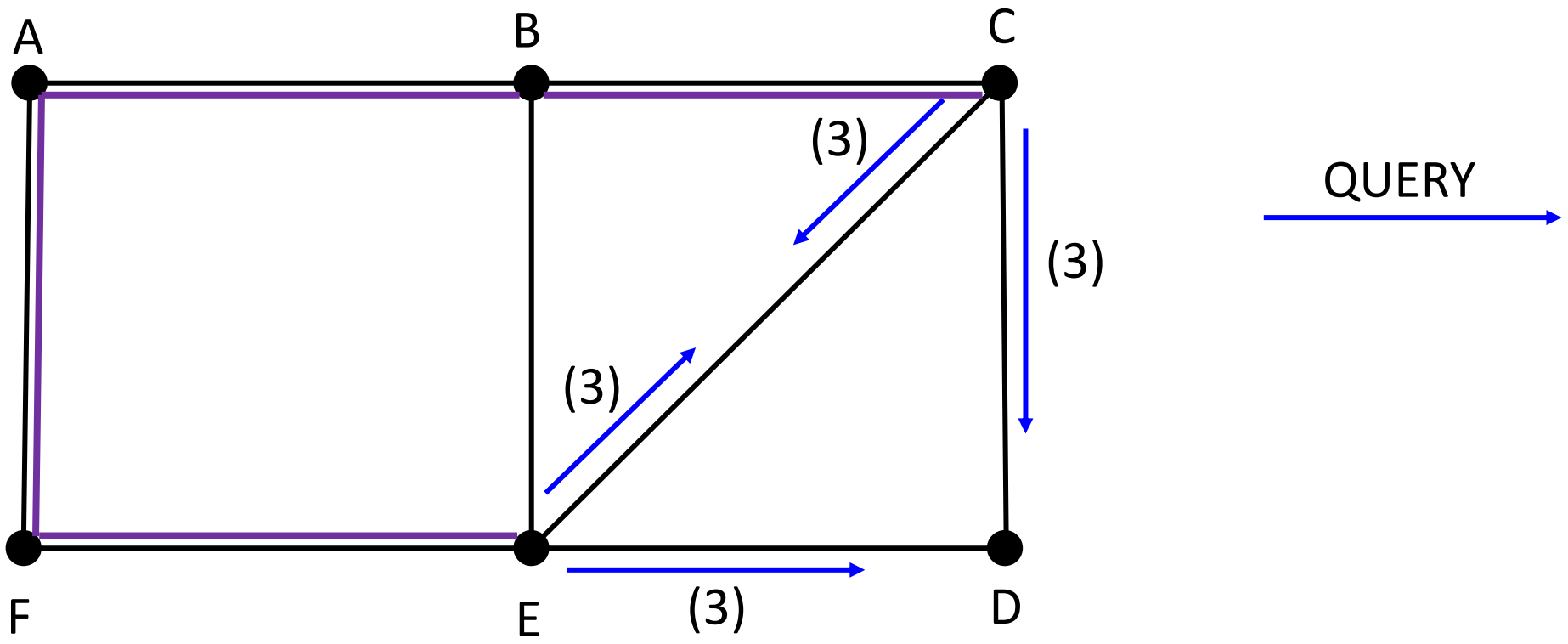
Synchronous Single-Initiator Spanning Tree Algorithm using Flooding



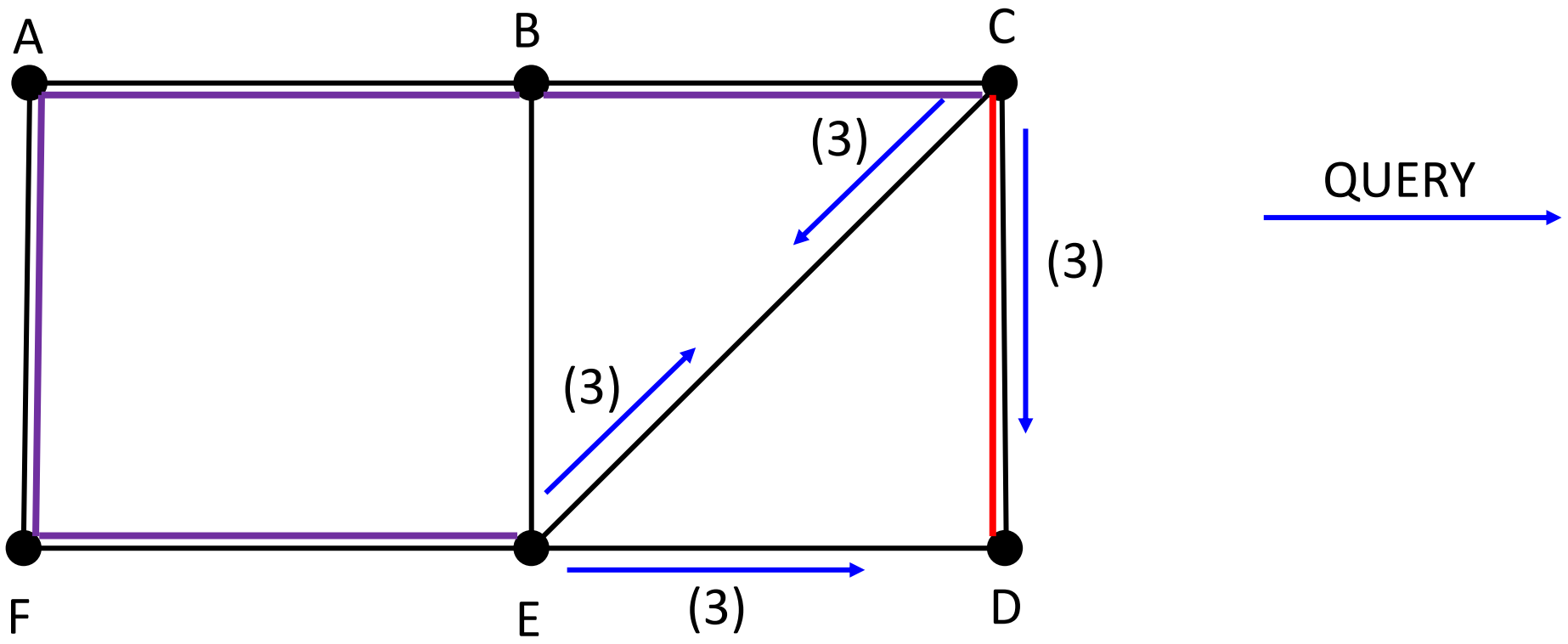
Synchronous Single-Initiator Spanning Tree Algorithm using Flooding



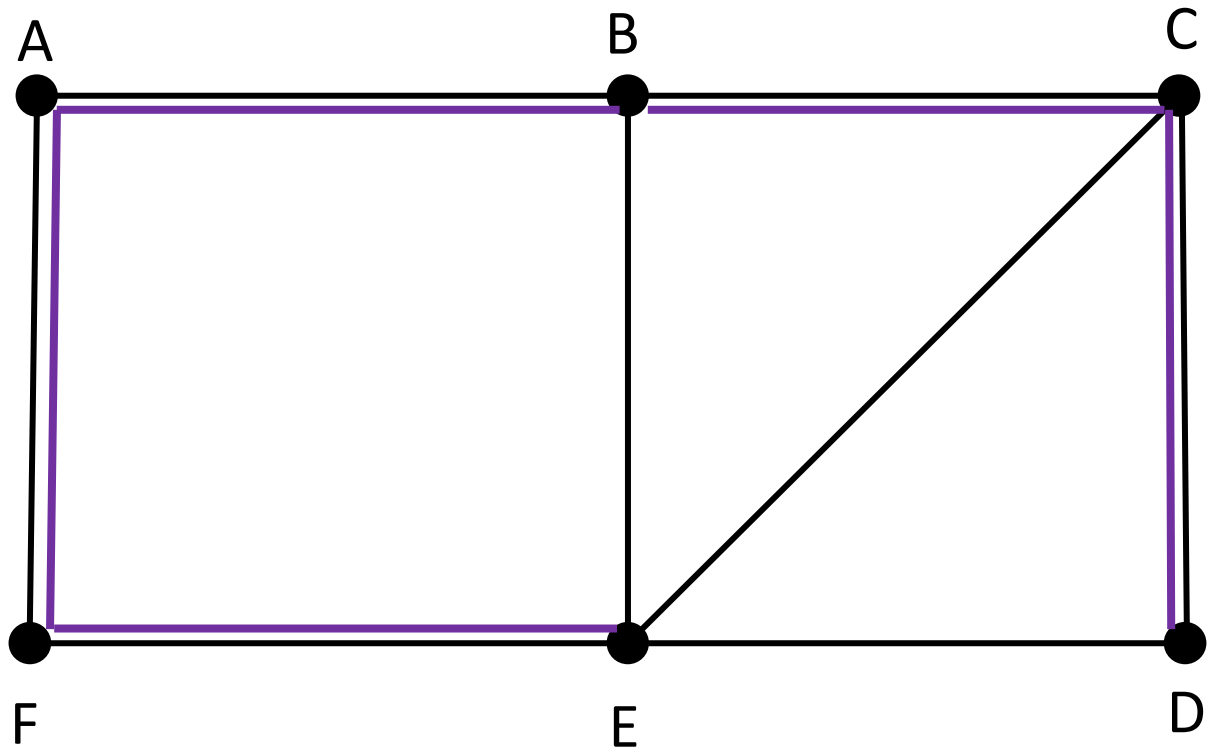
Synchronous Single-Initiator Spanning Tree Algorithm using Flooding



Synchronous Single-Initiator Spanning Tree Algorithm using Flooding



Synchronous Single-Initiator Spanning Tree Algorithm using Flooding

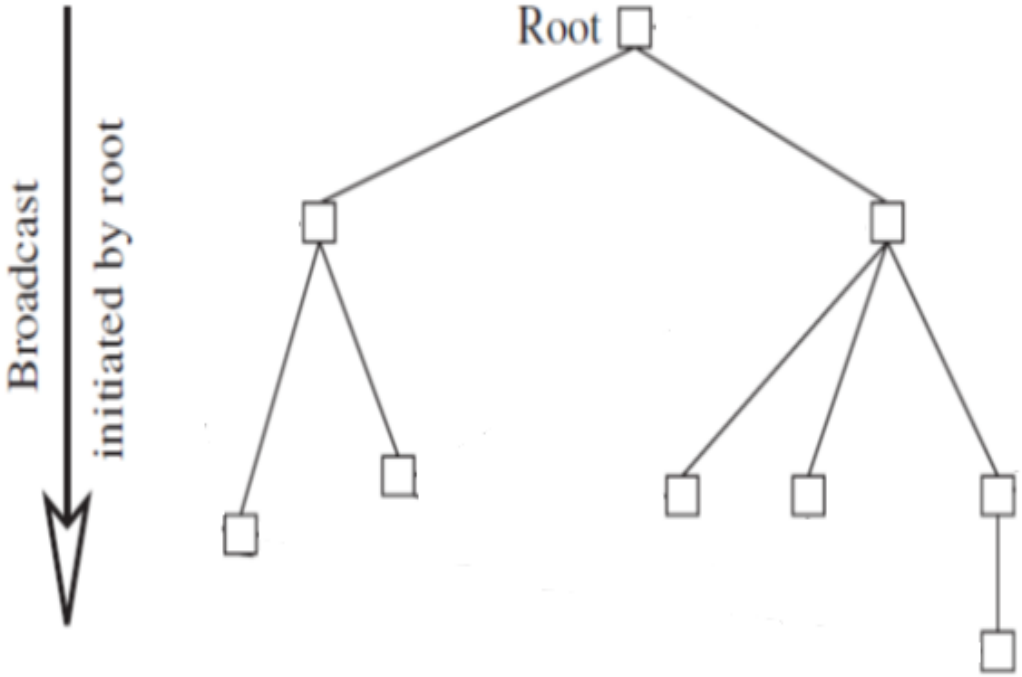


Broadcast and Convergecast on a Tree

- A spanning tree is useful for distributing (via a broadcast) and collecting (via a convergecast) information to/from all the nodes

Broadcast Algorithm:

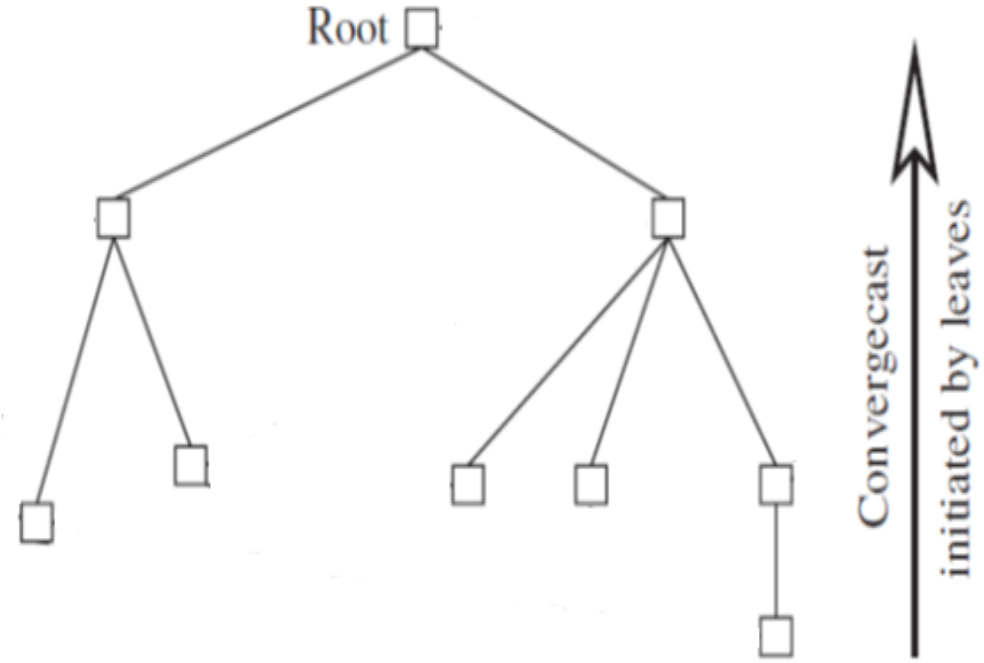
- ***BC1:***
  - The root sends the information to be broadcast to all its children.
- ***BC2:***
  - When a (non-root) node receives information from its parent, it copies it and forwards it to its children.



Broadcast and Convergecast on a Tree

Convergecast Algorithm:

- ***CVC1:***
  - Leaf node sends its report to its parent.
- ***CVC2:***
  - **At a non-leaf node that is not the root:** When a report is received from all the child nodes, the collective report is sent to the parent.
- ***CVC3:***
  - **At the root:** When a report is received from all the child nodes, the global function is evaluated using the reports.



Broadcast and Convergecast on a Tree

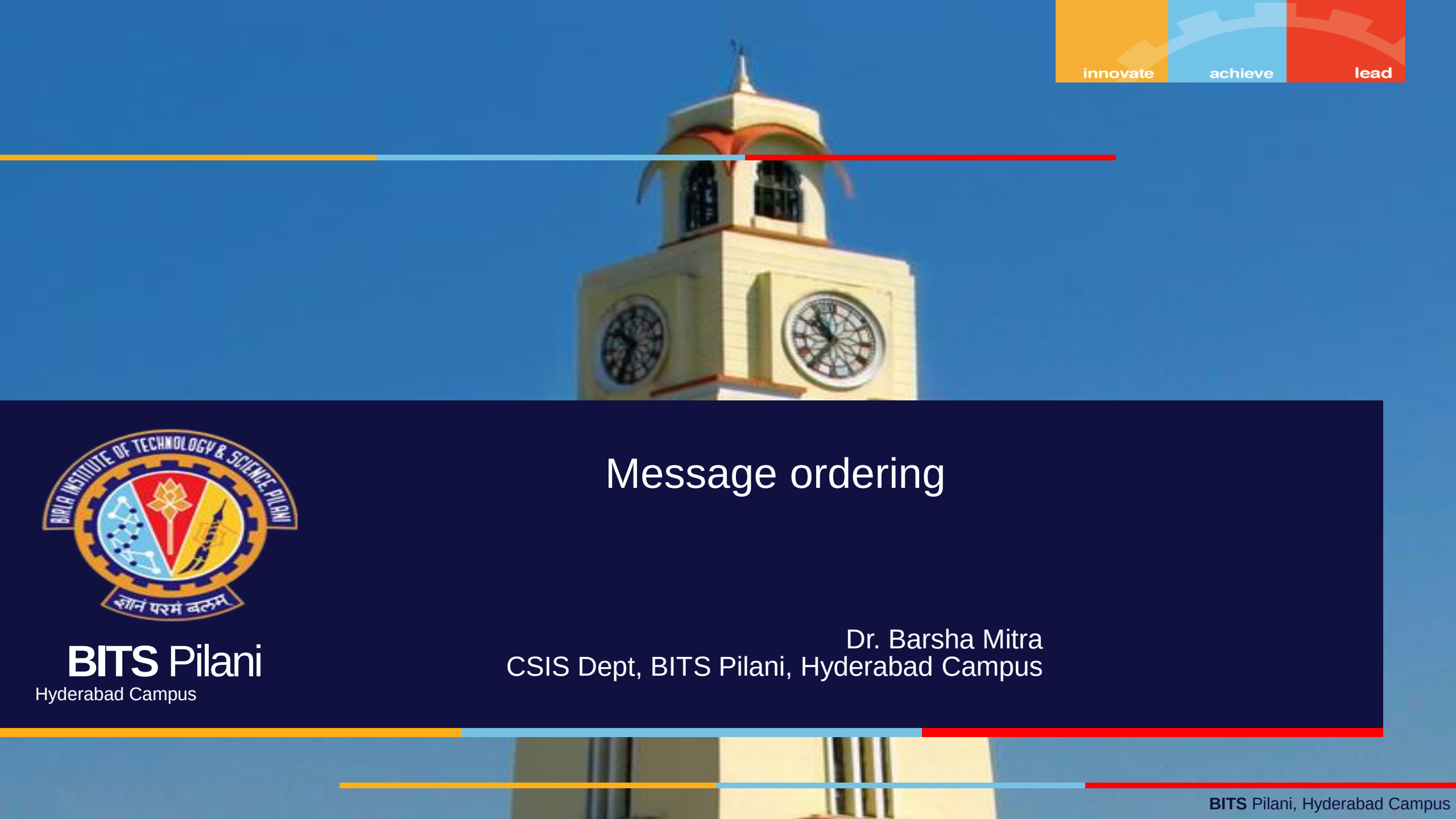
Complexity

- each broadcast and each convergecast requires $n - 1$ messages
- each broadcast and each convergecast requires time equal to the maximum height h of the tree, which is $O(n)$

Reference

- Ajay D. Kshemkalyani, and Mukesh Singhal, Chapter 5, “Distributed Computing: Principles, Algorithms, and Systems”, Cambridge University Press, 2008.

Thank You



innovate achieve lead

Message ordering

Dr. Barsha Mitra
CSIS Dept, BITS Pilani, Hyderabad Campus

BITS Pilani
Hyderabad Campus

Group Communication

- **broadcast** - sending a message to all members in the distributed system
- **multicasting** - a message is sent to a certain subset, identified as a group, of the processes in the system
- **unicasting** - point-to-point message communication

Causal Order

- A system supporting causal ordering model satisfies

  CO: for m_{ij} and m_{kj} , if send(m_{ij}) → send(m_{kj}), then rec(m_{ij}) → rec(m_{kj})

- 2 criteria must be satisfied by a causal ordering protocol:
  - Safety
  - Liveness

Causal Order

Safety

- a message M arriving at a process may need to be buffered until all system-wide messages sent in the causal past of the send(M) event to the same destination have already arrived
- distinction is made between
  - arrival of a message at a process
  - event at which the message is given to the application process

Liveness

A message that arrives at a process must eventually be delivered to the process

Raynal–Schiper–Toueg Algorithm

- each message M should carry a log of
  - all other messages
  - sent causally before M's send event, and sent to the same destination dest(M)
- log can be examined to ensure whether it is safe to deliver a message
- channels are FIFO

Raynal–Schiper–Toueg Algorithm

local variables:

- **array of int** SENT[1 …… n, 1 …… n] (n x n array)
  - $SENT_i[j, k]$ = no. of messages sent by P_j to P_k as known to P_i
- **array of int** DELIV [1 …… n]
  - $DELIV_i[j]$ = no. of messages from P_j that have been delivered to P_i

(1) **send event**, where P_i wants to send message M to P_j :

(1a) **send** (M, SENT) to P_j

(1b) SENT[i, j] = SENT[i, j] + 1

Raynal–Schiper–Toueg Algorithm

(2) **message arrival**, when (M, ST) arrives at P_i from P_j :

(2a) **deliver** M to P_i when for each process x,

(2b) $DELIV_i[x] \geq ST[x, i]$

(2c) $\forall$ x, y, $SENT[x, y] = \max(SENT[x, y], ST[x, y])$

(2d) $DELIV_i[j] = DELIV_i[j] + 1$

Raynal–Schiper–Toueg Algorithm

Complexity:

- space requirement at each process: $O(n^2)$ integers
- space overhead per message: n^2 integers
- time complexity at each process for each send and deliver event: $O(n^2)$

Raynal–Schiper–Toueg Algorithm

- P_1 sent 3 messages to P_2
- P_1 sent 4 messages to P_3
- P_2 sent 5 messages to P_1
- P_2 sent 2 messages to P_3
- P_3 sent 4 messages to P_2
- Now if,
  - P_1 sends m_1 to P_2

SENT of P_1

0	3	4
5	0	2
0	4	0

$DELIV_1 = [0\ \ 4\ \ 0]$

$DELIV_2 = [3\ \ 0\ \ 4]$

$DELIV_3 = [3\ \ 2\ \ 0]$

Updated SENT of P_1

0	4	4
5	0	2
0	4	0

SENT of P_2

0	3	4
5	0	2
0	4	0

Raynal–Schiper–Toueg Algorithm

- P_1 sent 3 messages to P_2
- P_1 sent 4 messages to P_3
- P_2 sent 5 messages to P_1
- P_2 sent 2 messages to P_3
- P_3 sent 4 messages to P_2
- Now if,
  - P_3 sends m_2 to P_2

SENT of P_3

0	3	4
5	0	2
0	4	0

$DELIV_1 = [0\ \ 4\ \ 0]$

$DELIV_2 = [4\ \ 0\ \ 4]$

$DELIV_3 = [3\ \ 2\ \ 0]$

Updated SENT of P_3

0	3	4
5	0	2
0	5	0

SENT of P_2

0	3	4
5	0	2
0	4	0

Raynal–Schiper–Toueg Algorithm

- P_1 sent 3 messages to P_2
- P_1 sent 4 messages to P_3
- P_2 sent 5 messages to P_1
- P_2 sent 2 messages to P_3
- P_3 sent 4 messages to P_2
- Now if,
  - P_1 sends m_3 to P_3

SENT of P_1

0	4	4
5	0	2
0	4	0

$DELIV_1 = [0\ \ 4\ \ 0]$

$DELIV_2 = [3\ \ 0\ \ 4]$

$DELIV_3 = [3\ \ 2\ \ 0]$

Updated SENT of P_1

0	4	5
5	0	2
0	4	0

Birman-Schiper-Stephenson Protocol

- C_i = vector clock of P_i
- $C_i[j]$ = j^{th} element of C_i

- tm = vector timestamp for message m, stamped after local clock is incremented

P_i sends a message m to P_j

- P_i increments $C_i[i]$

- P_i sets the timestamp tm = C_i for message m

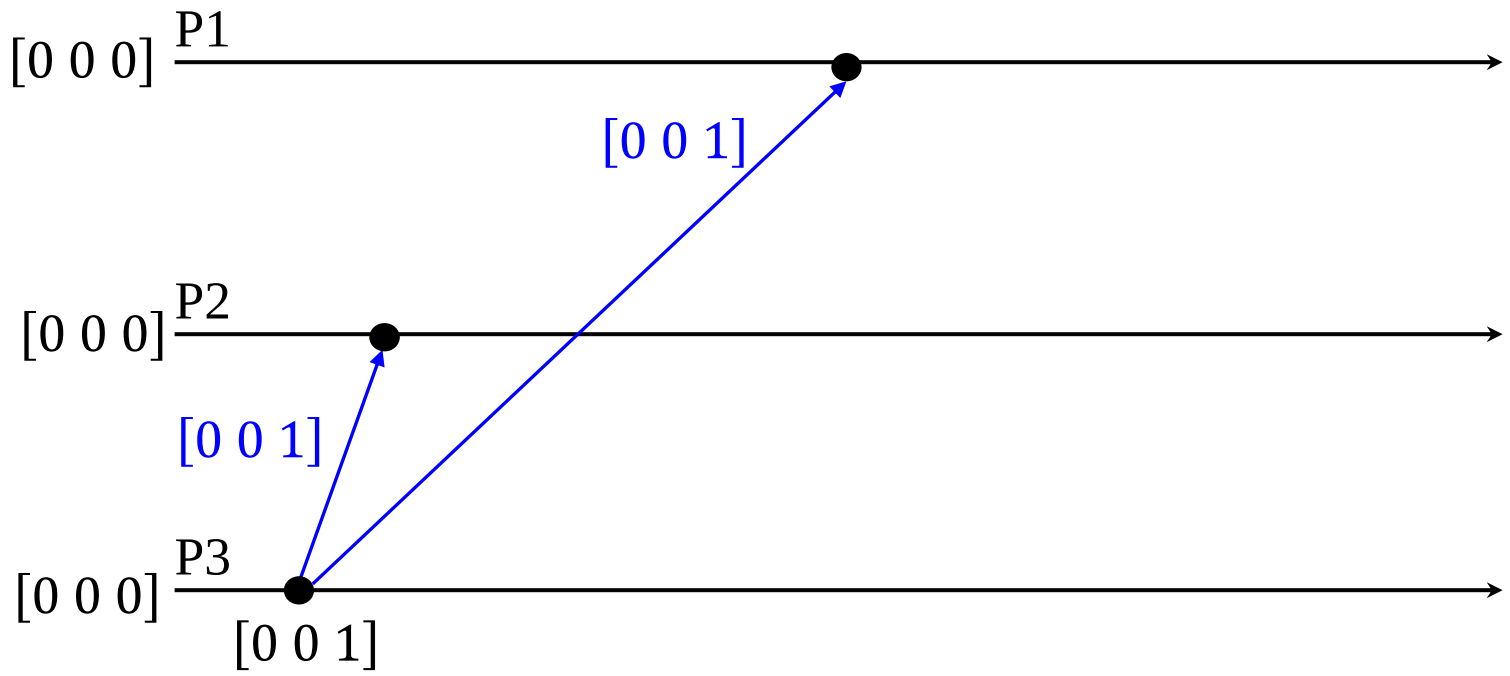
Birman-Schiper-Stephenson Protocol

P_j receives a message m from P_i

- when P_j ($j \neq i$) receives m with timestamp tm, it delays message delivery until:
  - $C_j[i] = tm[i] - 1$ //P_j has received all preceeding messages sent by P_i
  - $\forall$ k <= n and $k \neq i$, $C_j[k] \geq tm[k]$ //P_j has received all the messages that were received at P_i from other processes before P_i sent m
- when m is delivered to P_j, update P_j's vector clock

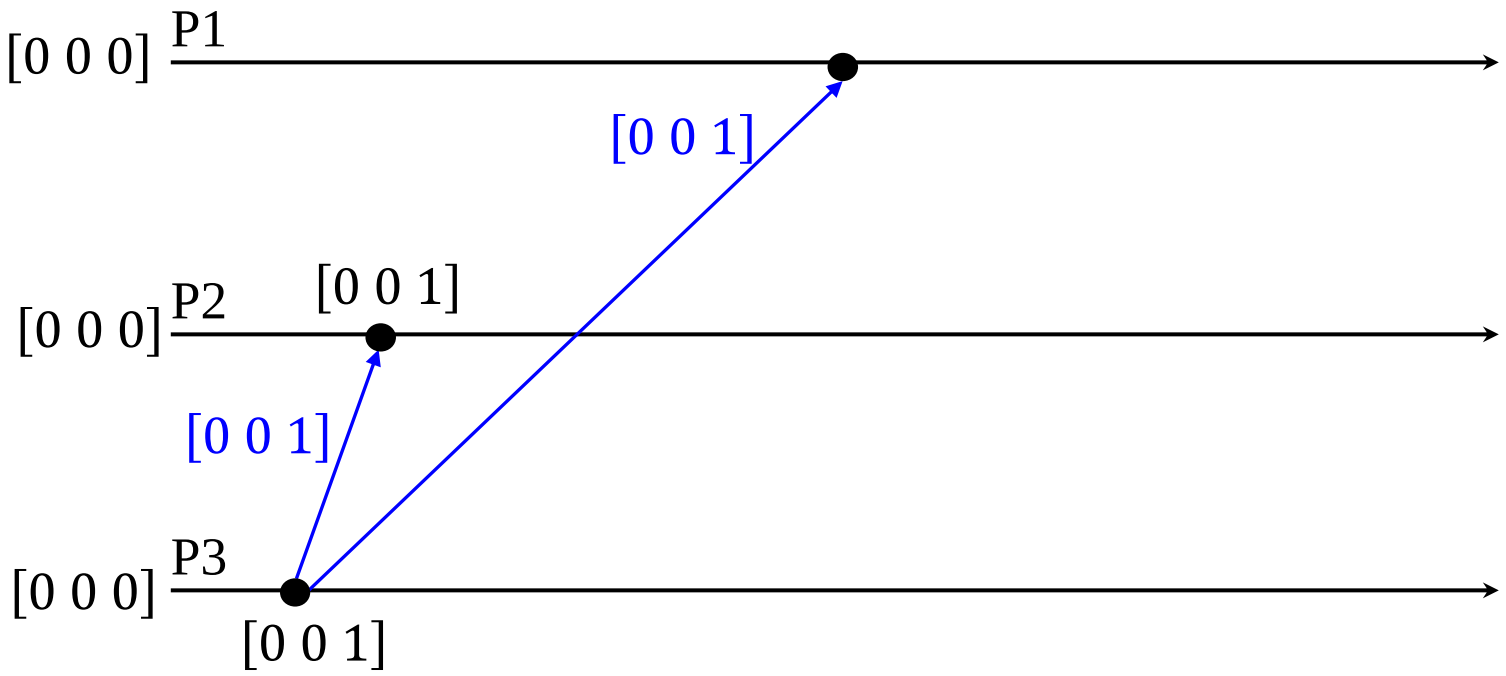
  $\forall i$, $C_j[i] = \max(C_j[i], tm[i])$
- check buffered messages to see if any can be delivered

Birman-Schiper-Stephenson Protocol

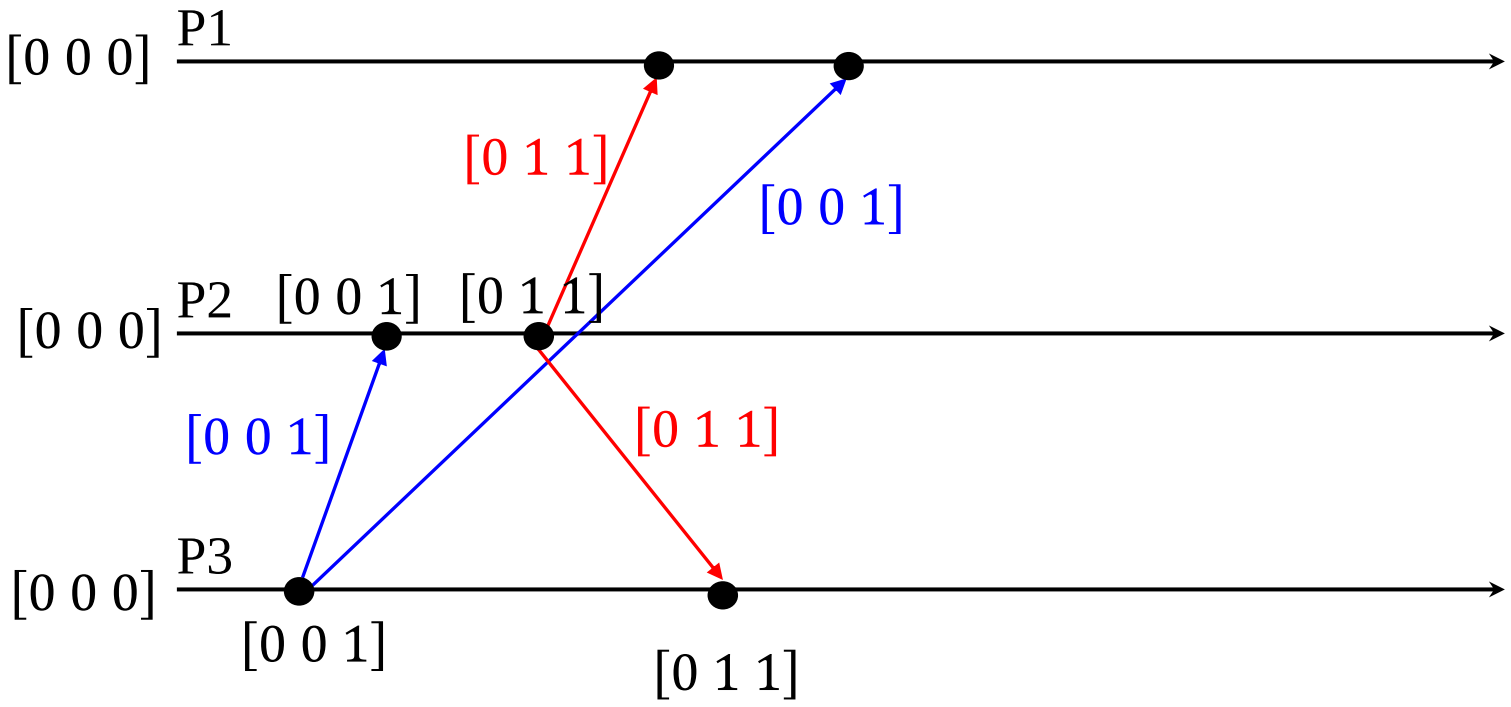


vectors in blue indicate timestamps of messages

Birman-Schiper-Stephenson Protocol

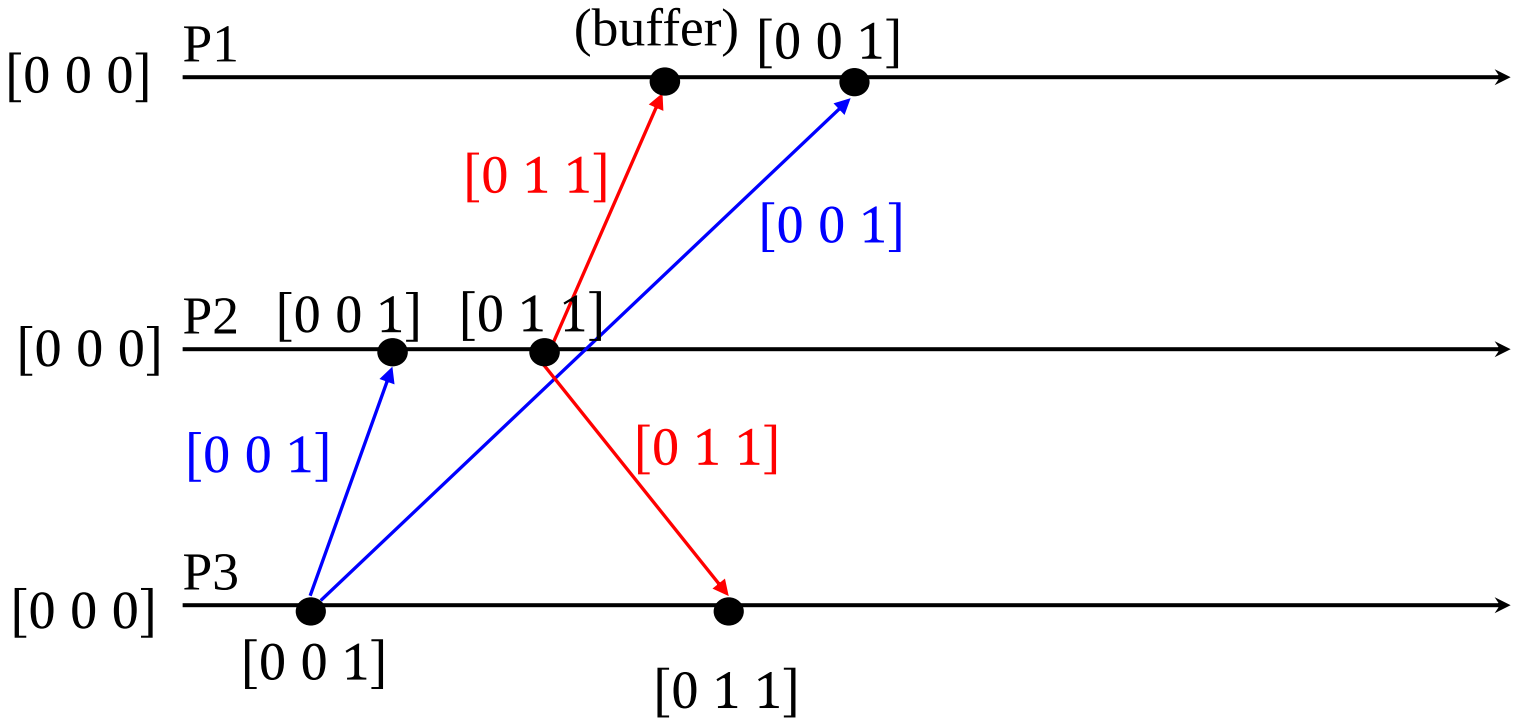


Birman-Schiper-Stephenson Protocol

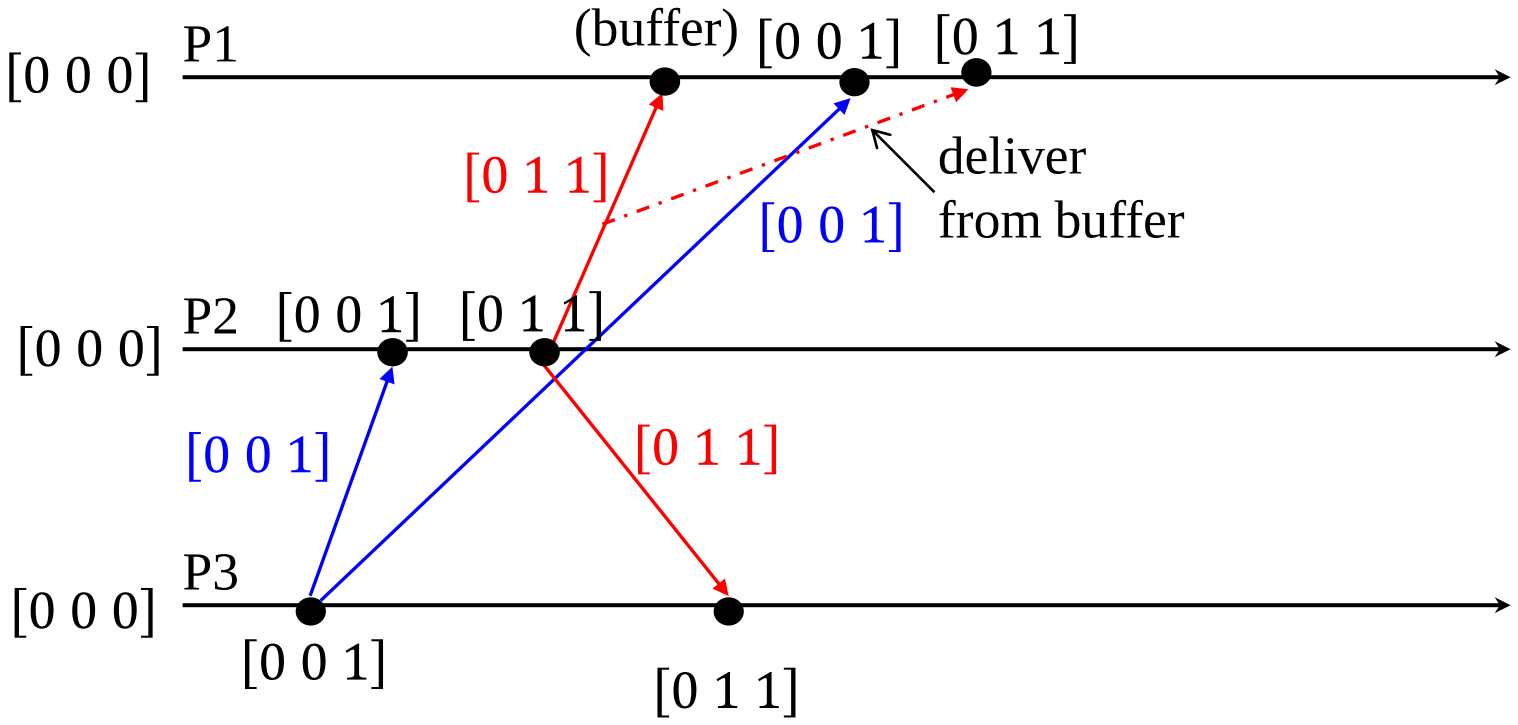


vectors in red indicate timestamps of messages

Birman-Schiper-Stephenson Protocol



Birman-Schiper-Stephenson Protocol

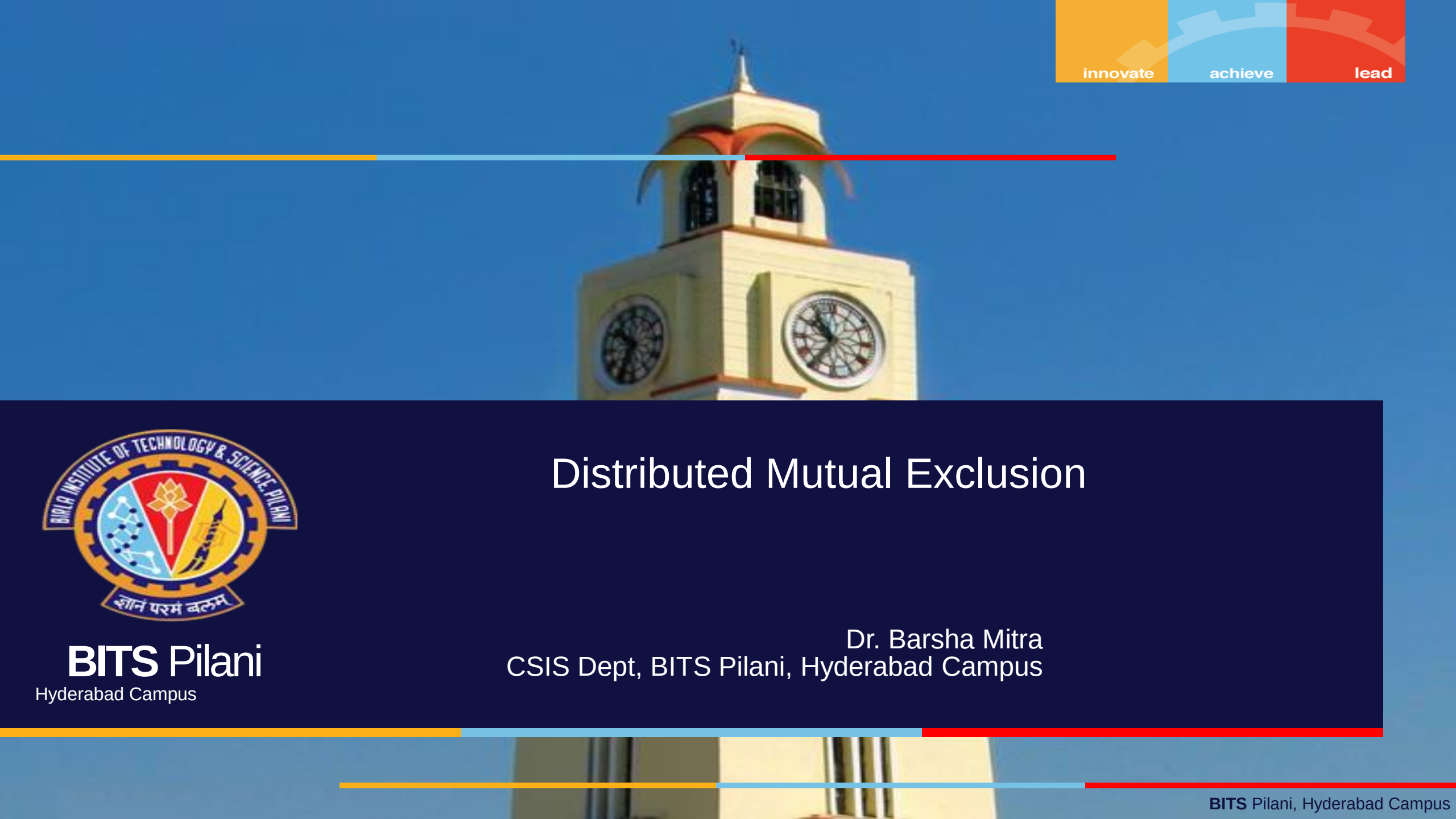


SES Protocol from Recorded Lecture

References

- Ajay D. Kshemkalyani, and Mukesh Singhal, Chapter 6, "Distributed Computing: Principles, Algorithms, and Systems", Cambridge University Press, 2008.
- Ajay D. Kshemkalyani, and Mukesh Singhal, Chapter 7, "Distributed Computing: Principles, Algorithms, and Systems", Cambridge University Press, 2008.
- https://courses.csail.mit.edu/6.006/fall11/rec/rec14.pdf

Thank You



innovate achieve lead

Distributed Mutual Exclusion

Dr. Barsha Mitra
CSIS Dept, BITS Pilani, Hyderabad Campus

BITS Pilani
Hyderabad Campus

Types of Approaches

- Token-based
- Assertion-based
- From Recorded Lecture
- Quorum-based

Quorum-Based Approach

- each site requests permission to execute the CS from a subset of sites called **quorum**
- quorums are formed in such a way that when two sites concurrently request access to the CS
  - at least one site receives both the requests
  - this site is responsible to make sure that only one request executes the CS at any time

Requirements of Mutual Exclusion Algorithms

- **Safety property**
  - at any instant, only one process can execute the CS
- **Liveness property**
  - a site must not wait indefinitely to execute the CS while other sites are repeatedly executing the CS
- **Fairness**
  - each process gets a fair chance to execute the CS
  - CS execution requests are executed in order of their arrival time
  - time is determined by a logical clock

Performance Metrics

- **From Recorded Lecture**

Lamport's Algorithm

- every site S_i keeps a queue, $request\_queue_i$
- $request\_queue_i$ contains mutual exclusion requests ordered by their timestamps
- FIFO
- CS requests are executed in increasing order of timestamps

Lamport's Algorithm

Requesting the critical section:

- When a site S_i wants to enter the CS, it broadcasts a REQUEST(ts_i, i) message to all other sites and places the request on $request\_queue_i$. ((ts_i, i) denotes the timestamp of the request)
- When a site S_j receives the REQUEST(ts_i, i) message from site S_i, it places site S_i's request on $request\_queue_j$ and returns a timestamped REPLY message to S_i

Lamport's Algorithm

Executing the critical section:

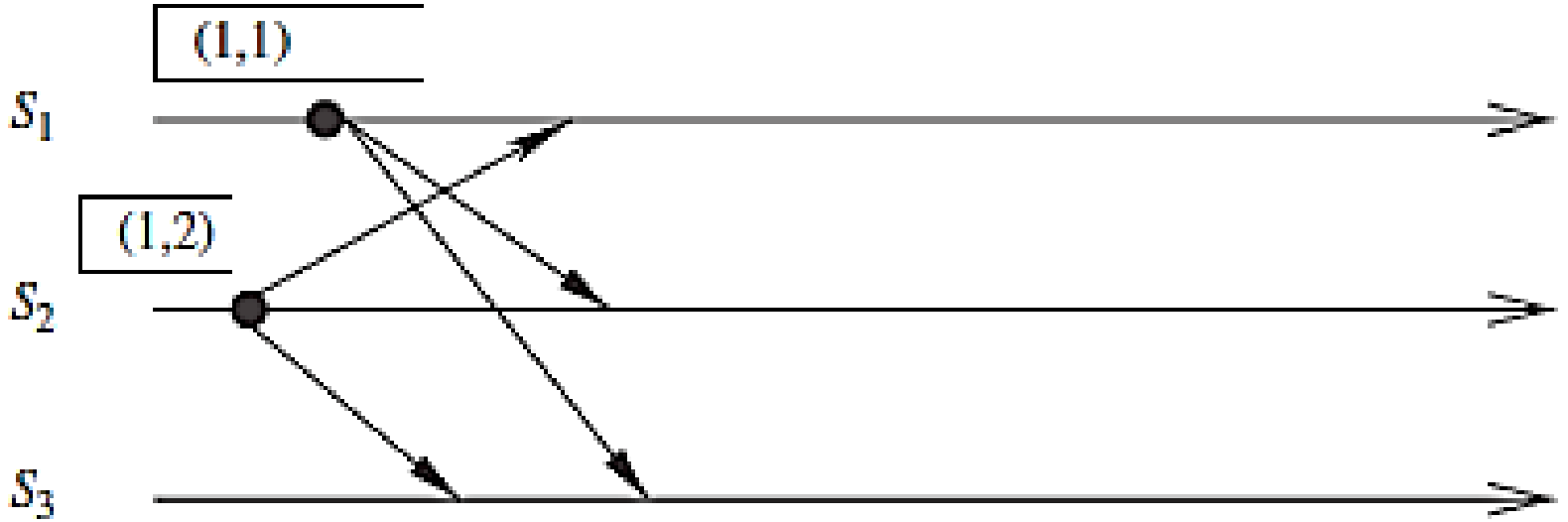
Site S_i enters the CS when the following two conditions hold:

- **L1**: S_i has received a message with timestamp larger than (ts_i, i) from all other sites
- **L2**: S_i's request is at the top of $request\_queue_i$

Releasing the critical section:

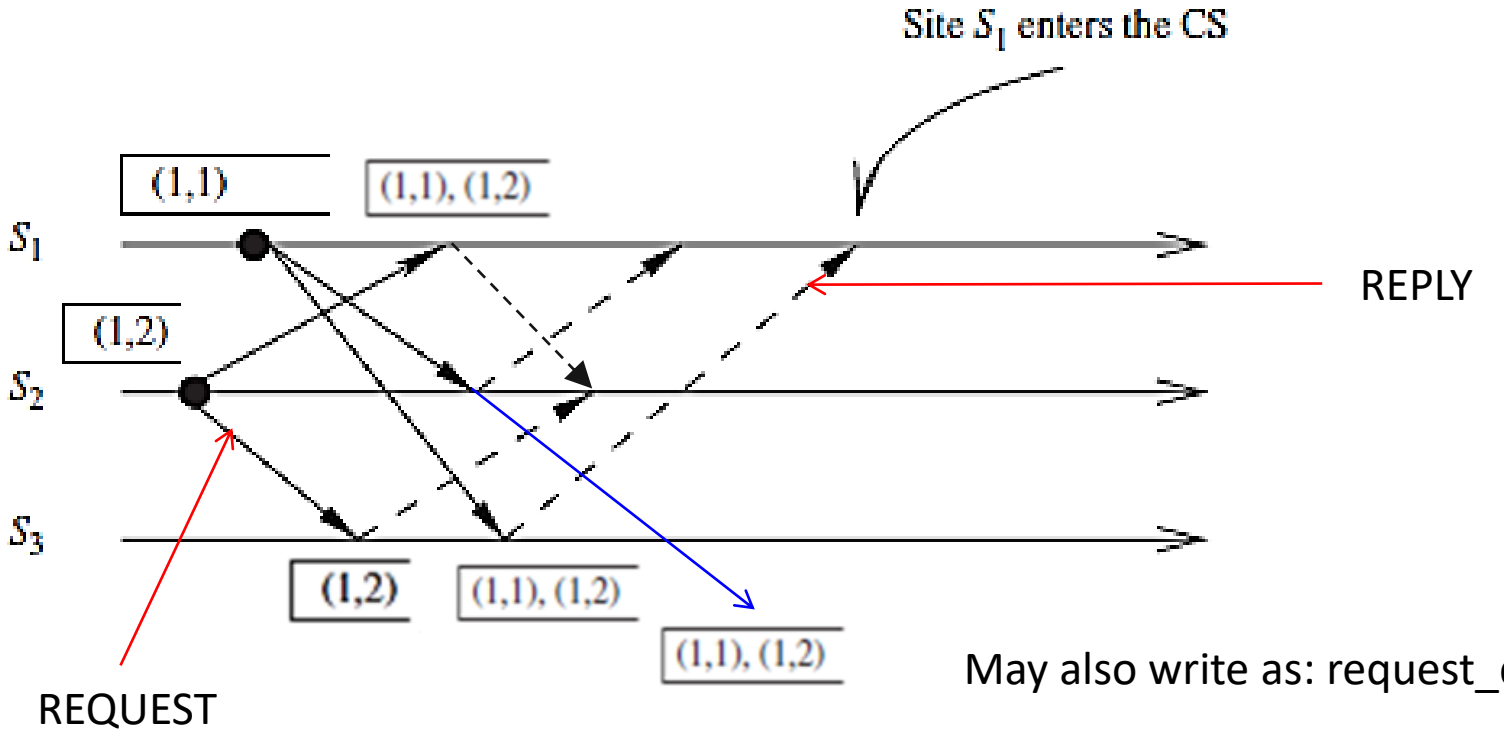
- Site S_i, upon exiting the CS, removes its request from the top of its request queue and broadcasts a timestamped RELEASE message to all other sites
- When a site S_j receives a RELEASE message from site S_i, it removes S_i's request from its request queue

Lamport's Algorithm



Sites S_1 and S_2 make requests for the CS

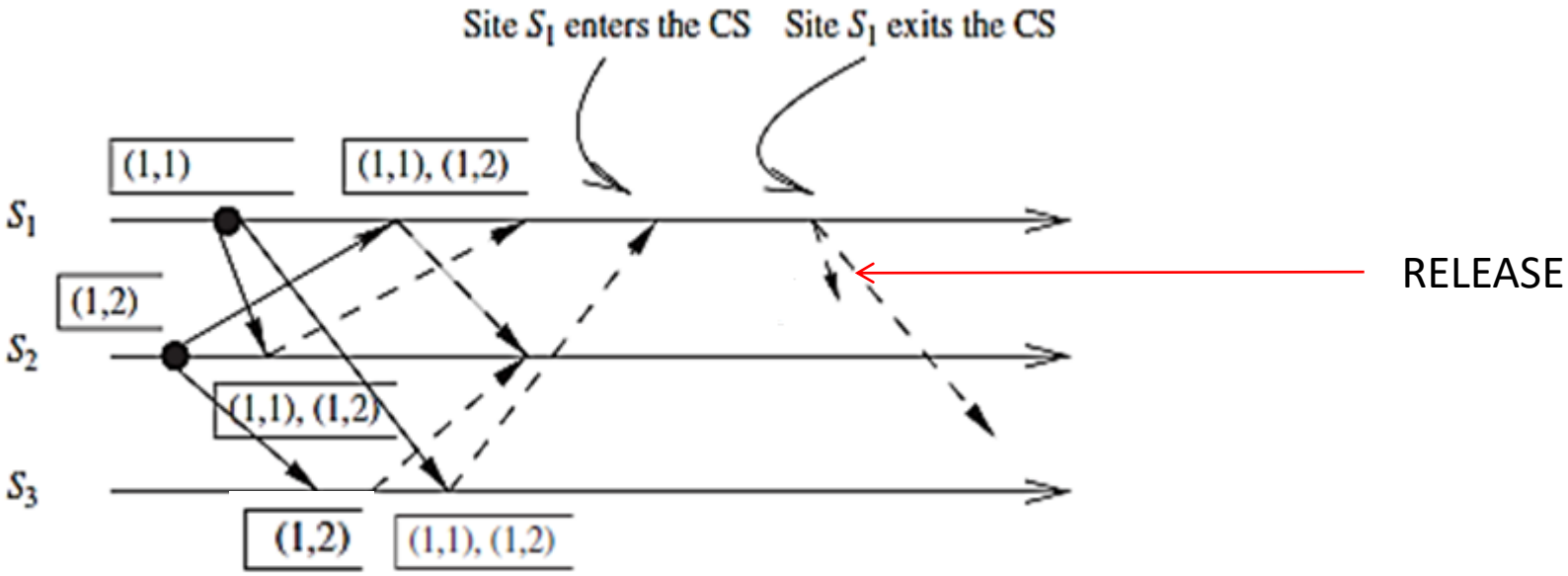
Lamport's Algorithm



May also write as: request_queue$_1$ = ((1, 1), (1, 2))

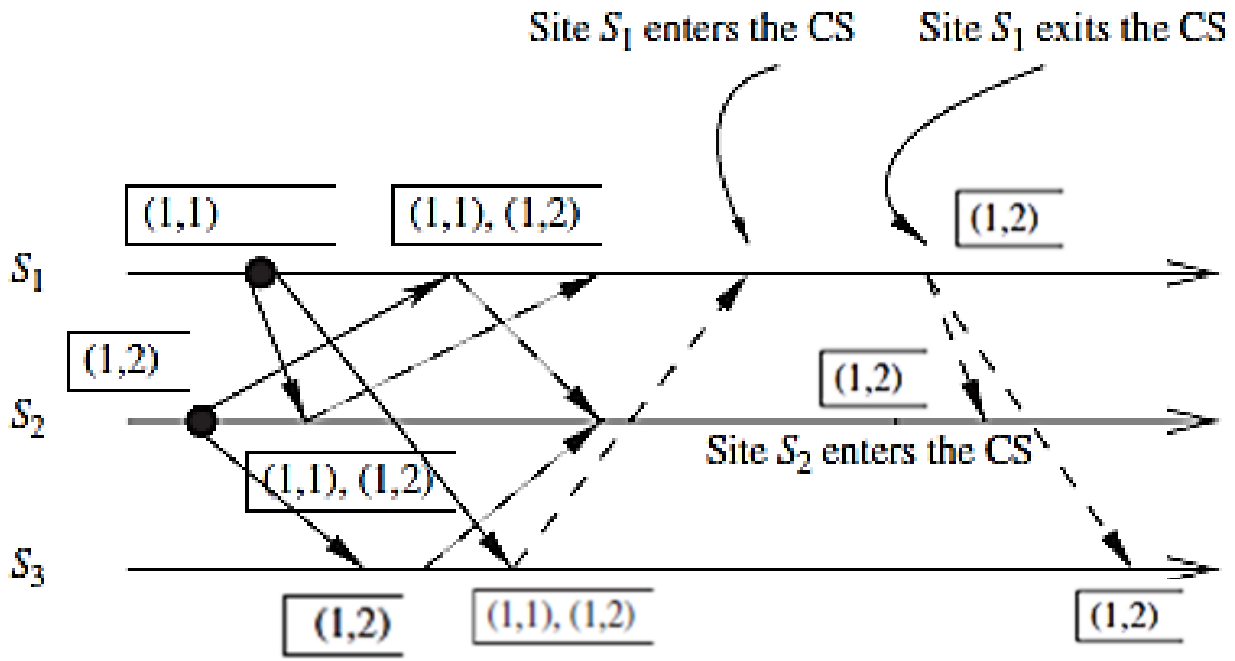
Site S_1 enters the CS

Lamport's Algorithm



Site S_1 exits the CS and sends RELEASE messages

Lamport's Algorithm



Site S_2 enters the CS

Performance - Requires 3(N – 1) messages per CS invocation

Ricart–Agrawala Algorithm

- communication channels are not required to be FIFO
- REQUEST and REPLY messages
- each process p_i maintains the request-deferred array, RD_i
- size of RD_i = no. of processes in the system
- initially, $\forall i \forall j$: $RD_i[j] = 0$
- whenever p_i defers the request sent by p_j, it sets $RD_i[j] = 1$,
- after it has sent a REPLY message to p_j , it sets $RD_i[j] = 0$

Ricart–Agrawala Algorithm

Requesting the critical section:

(a) When a site S_i wants to enter the CS, it broadcasts a timestamped REQUEST message to all other sites

(b) When site S_j receives a REQUEST message from site S_i, it sends a REPLY message to site S_i if site S_j is neither requesting nor executing the CS, or if the site S_j is requesting and S_i's request's timestamp is smaller than site S_j's own request's timestamp. Otherwise, the reply is deferred and S_j sets RD_j [i] = 1

Ricart–Agrawala Algorithm

Executing the critical section:

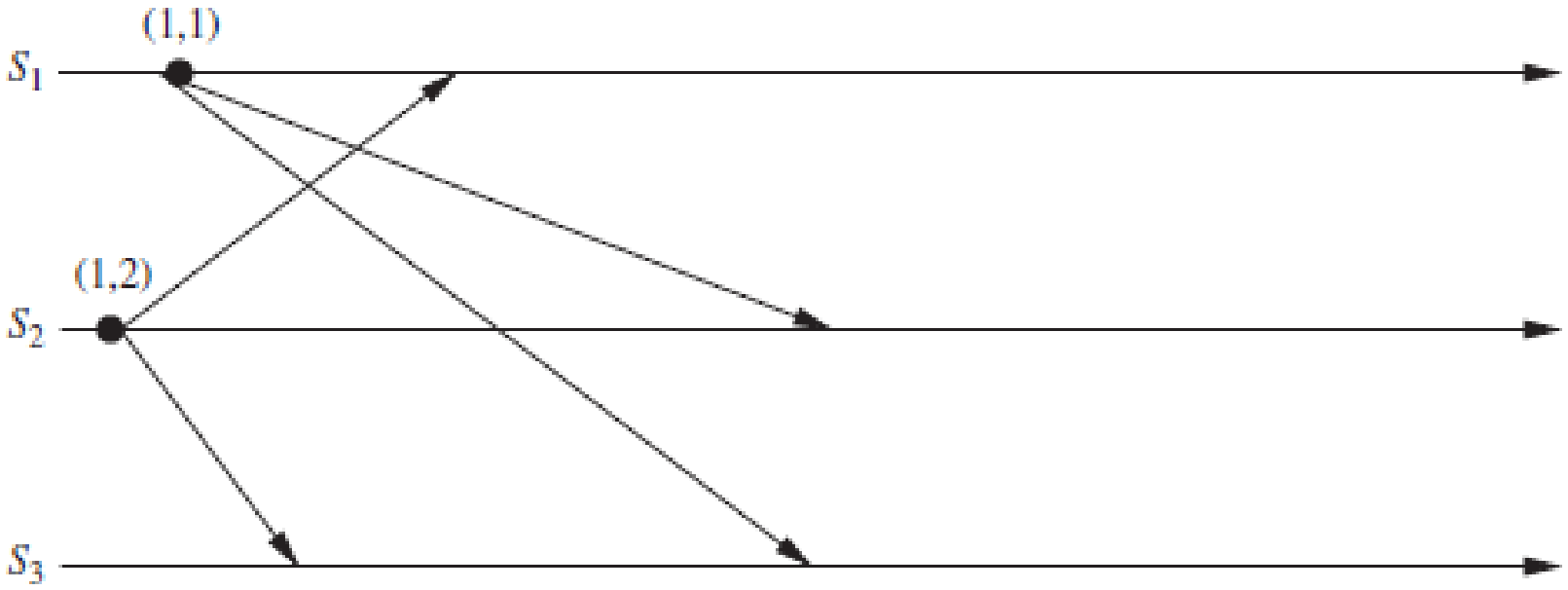
(c) Site S_i enters the CS after it has received a REPLY message from every site it sent a REQUEST message to

Releasing the critical section:

(d) When site S_i exits the CS, it sends all the deferred REPLY messages:

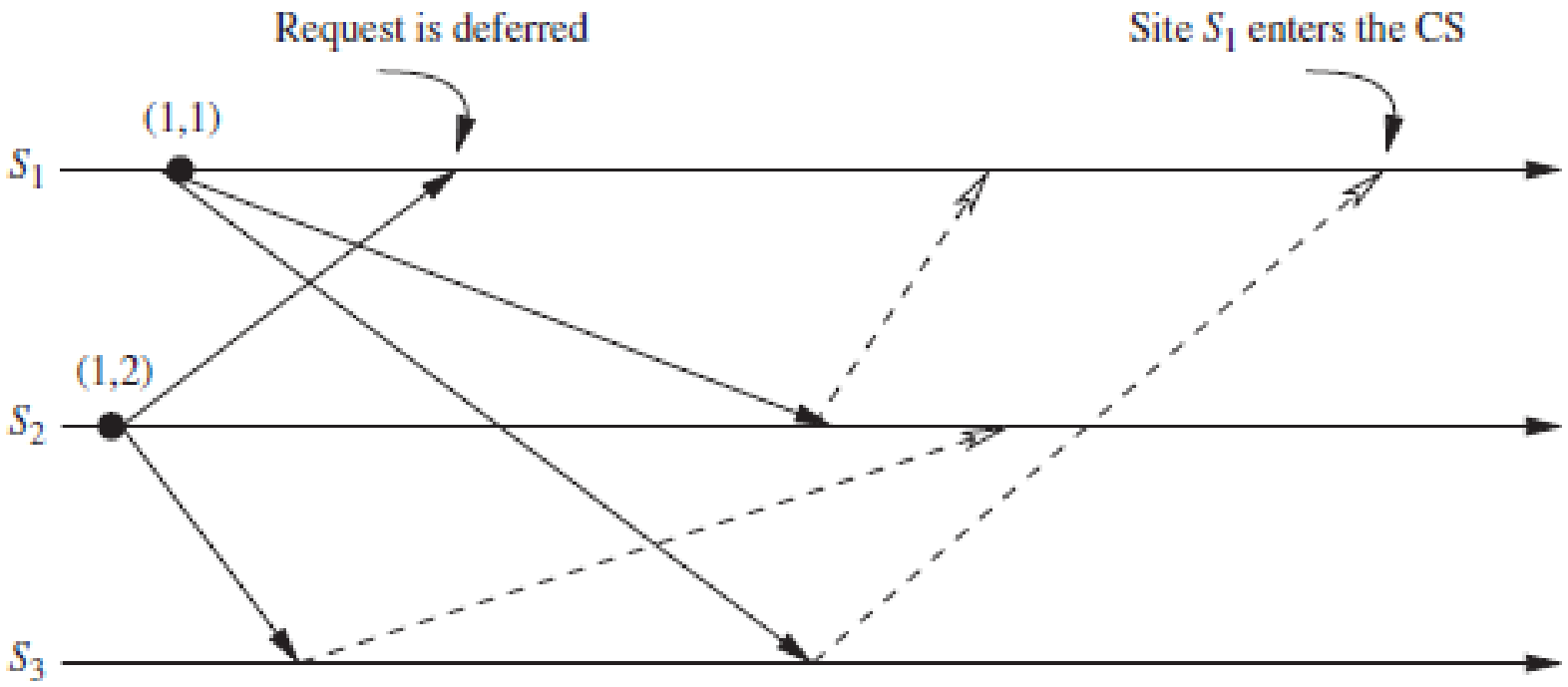
$\forall j$ if $RD_i[j] = 1$, then S_i sends a REPLY message to S_j and sets $RD_i[j] = 0$

Ricart–Agrawala Algorithm



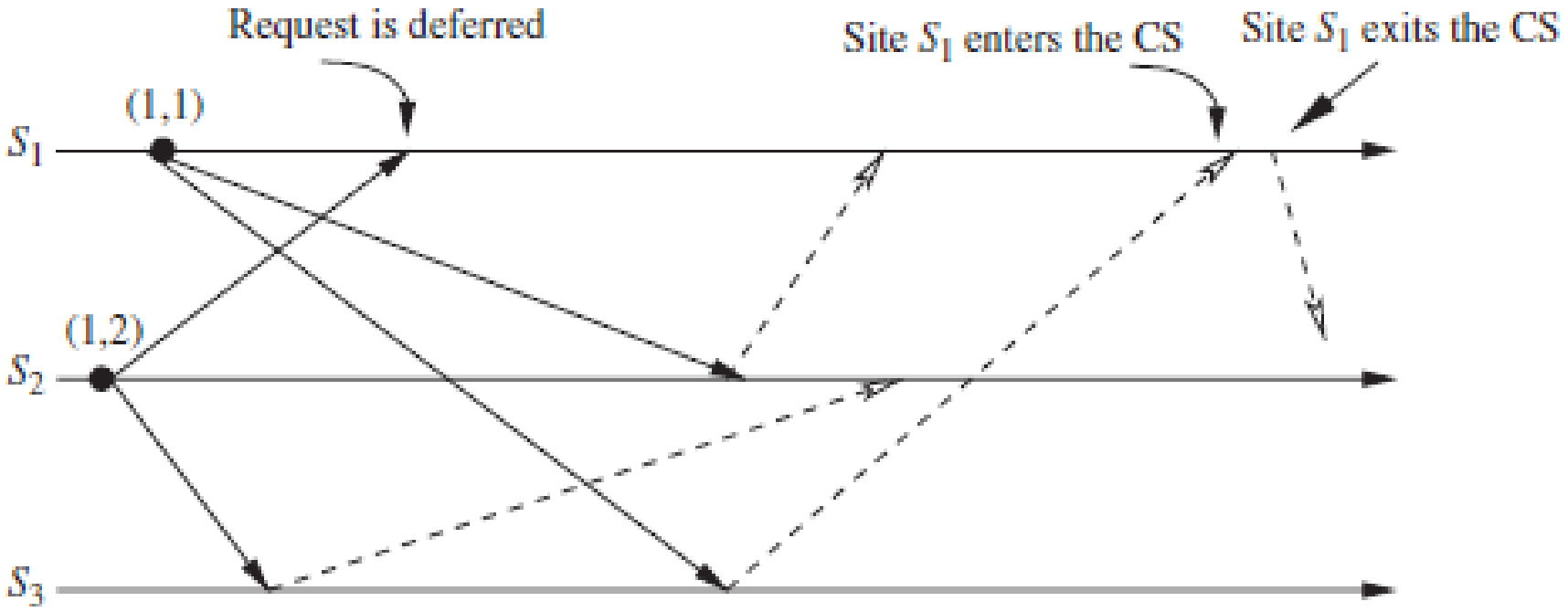
Sites S_1 and S_2 each make a request for the CS

Ricart–Agrawala Algorithm



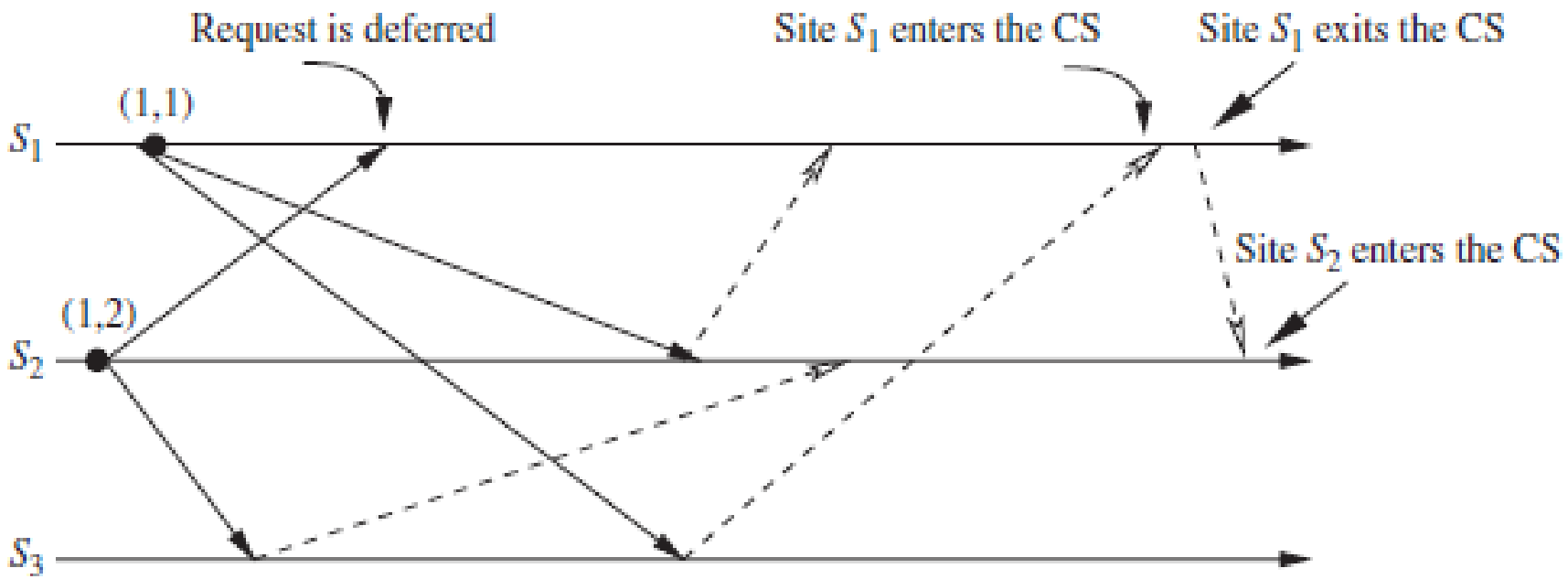
Site S_1 enters the CS

Ricart–Agrawala Algorithm



Site S_1 exits the CS and sends a REPLY message to S_2's deferred request

Ricart–Agrawala Algorithm



Site S_2 enters the CS

Performance - requires 2(N − 1) messages per CS execution

Maekawa's Algorithm

- quorum-based mutual exclusion algorithm
- request sets for sites (i.e., quorums) are constructed to satisfy the following conditions:
  - M1: $(\forall i\ \forall j : i \neq j, 1 \leq i, j \leq N :: R_i \cap R_j \neq \phi)$
  - M2: $(\forall i : 1 \leq i \leq N :: S_i \in R_i)$
  - M3: $(\forall i : 1 \leq i \leq N :: |R_i| = K$ for some $K)$
  - M4: Any site S_j is contained in K number of R_i's, $1 \leq i, j \leq N$
- Maekawa showed that $N = K(K - 1) + 1$
- This relation gives $|R_i| = K = \sqrt{N}$ (square root of N)
- Uses REQUEST, REPLY and RELEASE messages
- Performance → $3\sqrt{N}$ messages per CS invocation

Maekawa's Algorithm: Problem

Maekawa's Algorithm

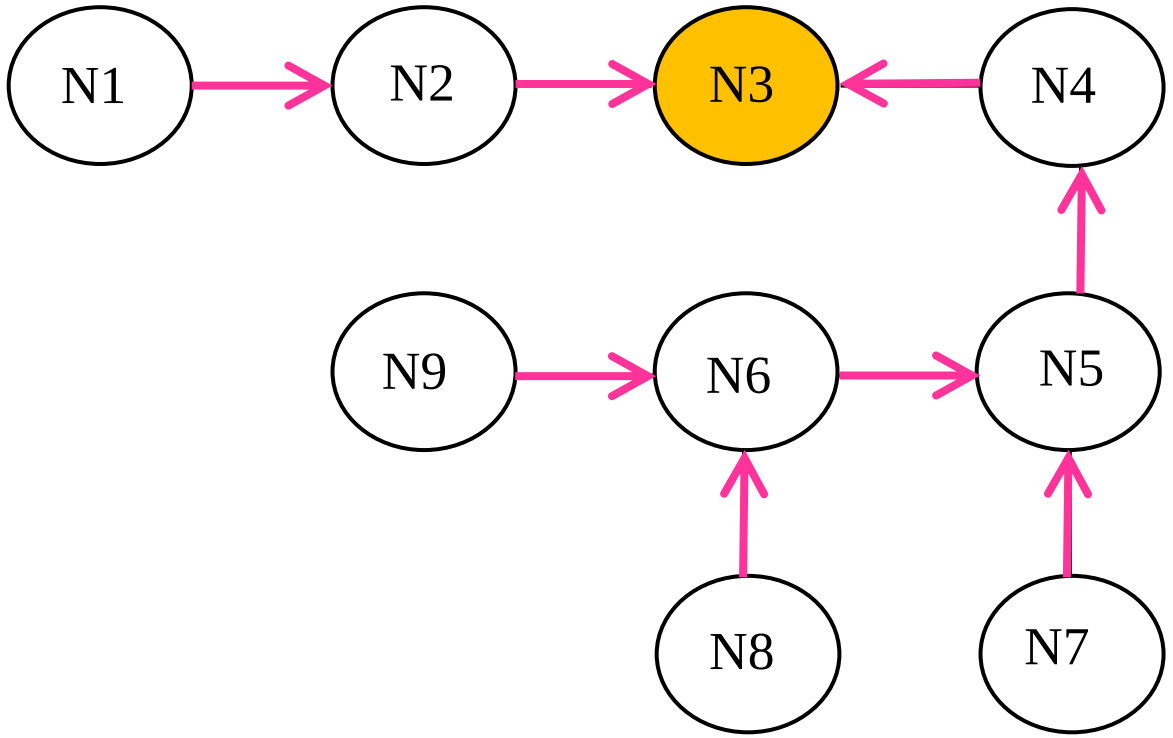
- Algorithm Steps: from Recorded Lecture
- Deadlocks: from Recorded Lecture

Raymond's Tree-Based Algorithm

- uses a spanning tree of the network
- each node maintains a HOLDER variable that provides information about the placement of the privilege in relation to the node itself
- a node stores in its HOLDER variable the identity of a node that it thinks has the privilege or leads to the node having the privilege
- for 2 nodes X and Y, if $HOLDER_X$ = Y, the undirected edge between X and Y can be redrawn as a directed edge from X to Y
- Node containing the privilege is also known as the root node

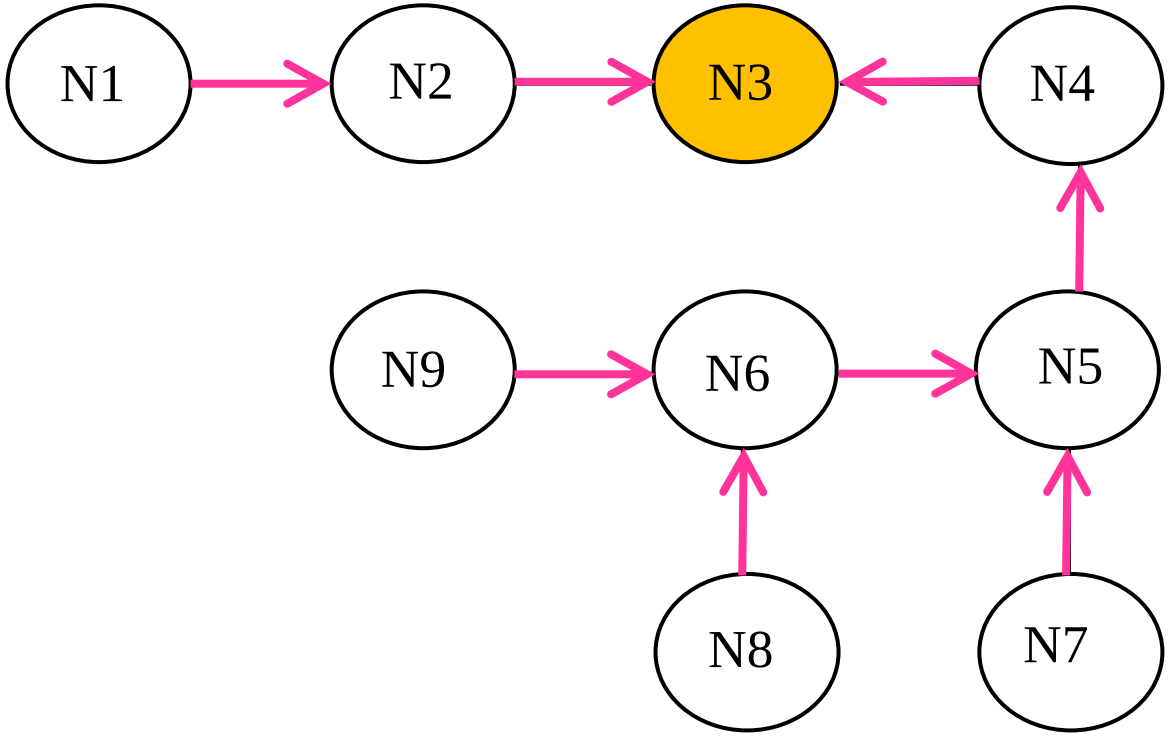
Raymond's Tree-Based Algorithm

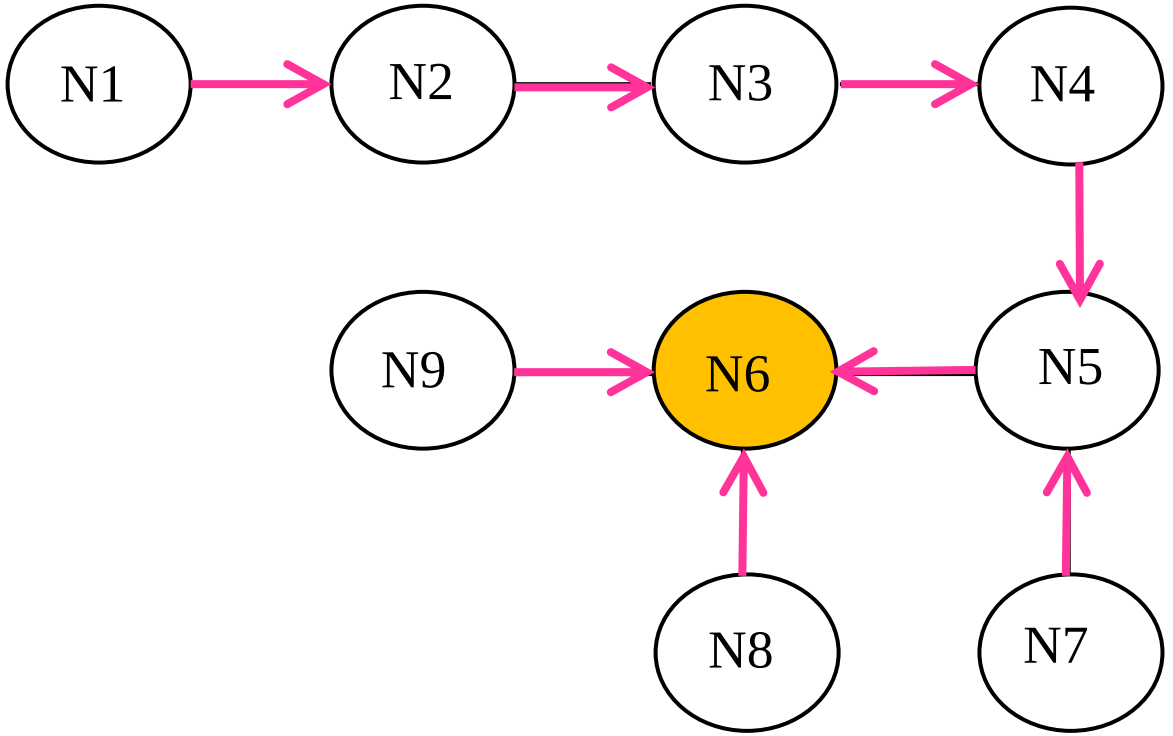
- messages between nodes traverse along undirected edges of the tree
- node needs to hold information about and communicate only to its immediate-neighboring nodes



- $HOLDER_{N1}$ = N2
- $HOLDER_{N2}$ = N3
- $HOLDER_{N3}$ = self
- $HOLDER_{N4}$ = N3
- $HOLDER_{N5}$ = N4
- $HOLDER_{N6}$ = N5
- $HOLDER_{N7}$ = N5
- $HOLDER_{N8}$ = N6
- $HOLDER_{N9}$ = N6

Raymond's Tree-Based Algorithm





Raymond's Tree-Based Algorithm

Data Structures

- HOLDER
- USING
  - possible values true or false
  - indicates if the current node is executing the critical section
- ASKED
  - possible values true or false
  - indicates if node has sent a request for the privilege
  - prevents the sending of duplicate requests for privilege

Raymond's Tree-Based Algorithm

Data Structures

- REQUEST_Q
  - FIFO queue that can contain "self " or the identities of immediate neighbors as elements
  - REQUEST_Q of a node consists of the identities of those immediate neighbors that have requested for privilege but have not yet been sent the privilege
  - maximum size of REQUEST_Q of a node is the number of immediate neighbors + 1 (for "self ")

Suzuki–Kasami's Broadcast Algorithm

- broadcasts a REQUEST message for the token to all other sites
- site that possesses the token sends it to the requesting site upon the receipt of its REQUEST message
- if a site receives a REQUEST message when it is executing the CS, it sends the token only after it has completed the CS execution

Suzuki–Kasami's Broadcast Algorithm

- Distinguish outdated and current REQUEST messages
- REQUEST(j, sn) where sn (sn = 1, 2…..) is a sequence number that indicates that S_j is requesting its sn^{th} CS execution
- S_i keeps an array of integers $RN_i[1, \ldots, n]$ where $RN_i[j]$ is the largest sequence number received in a REQUEST message so far from S_j
- when S_i receives a REQUEST(j, sn) message, it sets $RN_i[j] = \max(RN_i[j], sn)$
- when S_i receives a REQUEST(j,sn) message, the request is outdated if $RN_i[j] > sn$

Suzuki–Kasami's Broadcast Algorithm

- sites with outstanding requests for the CS are determined in the following manner:
  - token consists of a queue of requesting sites, Q, and an array of integers LN[1, … ,n], where LN[j] is the sequence number of the request which S_j executed most recently
  - after executing its CS, a S_i updates $LN[i] = RN_i[i]$
  - token array LN[1, … ,n] permits a site to determine if a site has an outstanding request for the CS
  - at S_i, if $RN_i[j] = LN[j] + 1$, then S_j is currently requesting a token
  - after executing the CS, a site checks this condition for all the j's to determine all the sites that are requesting the token and places their i.d.'s in queue Q if these i.d.'s are not already present in Q
  - finally, the site sends the token to the site whose i.d. is at the head of Q

Suzuki–Kasami's Broadcast Algorithm

Requesting the critical section:

(a) If requesting site S_i does not have the token, then it increments its sequence number, $RN_i[i]$, and sends a REQUEST(i, sn) message to all other sites. ("sn" is the updated value of $RN_i[i]$.)

(b) When a site S_j receives this message, it sets $RN_j[i]$ to max($RN_j[i]$, sn). If S_j has the idle token, then it sends the token to S_i if $RN_j[i] = LN[i]+1$.

Executing the critical section:

(c) Site S_i executes the CS after it has received the token.

Suzuki–Kasami's Broadcast Algorithm

Releasing the critical section: Having finished the execution of the CS, site S_i takes the following actions:
(d) It sets LN[i] element of the token array equal to $RN_i[i]$.

(e) For every site S_j whose i.d. is not in the token queue, it appends its i.d. to the token queue if $RN_i[j] = LN[j] + 1$.

(f) If the token queue is non-empty after the above update, S_i deletes the top site i.d. from the token queue and sends the token to the site indicated by the i.d.

Suzuki–Kasami's Broadcast Algorithm

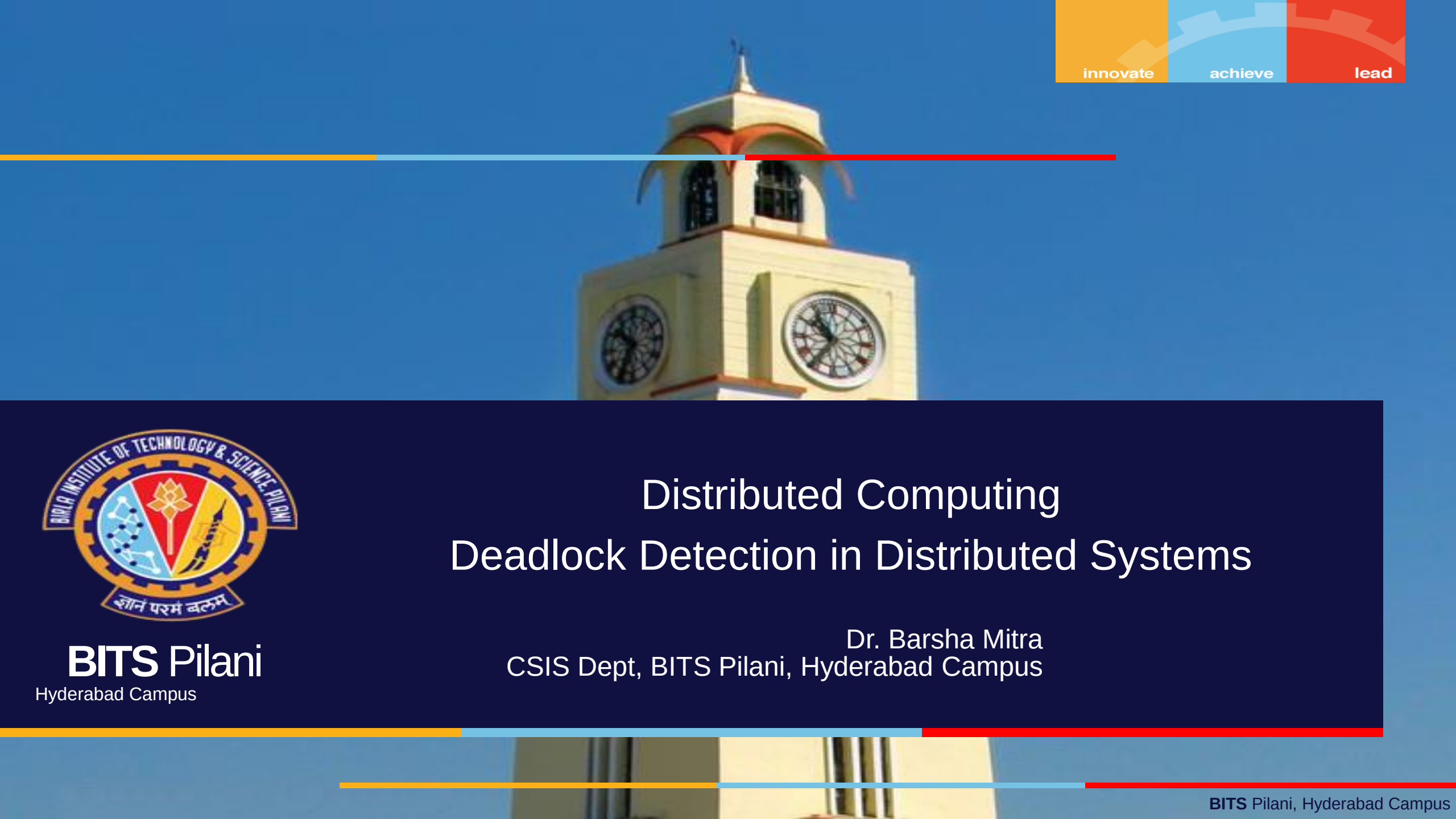
Performance

- if a site holds the token, no message is required
- if a site does not hold the token, N messages are required

Reference

- Ajay D. Kshemkalyani, and Mukesh Singhal, Chapter 9, “Distributed Computing: Principles, Algorithms, and Systems”, Cambridge University Press, 2008.

Thank You



innovate achieve lead

BITS Pilani
Hyderabad Campus

Distributed Computing
Deadlock Detection in Distributed Systems

Dr. Barsha Mitra
CSIS Dept, BITS Pilani, Hyderabad Campus

Introduction

- a process may request resources in any order
- request order may not be known apriori
- a process can request a resource while holding others
- if the allocation sequence of process resources is not controlled in such environments
  - deadlocks can occur
- **deadlock** - condition where a set of processes request resources that are held by other processes in the set

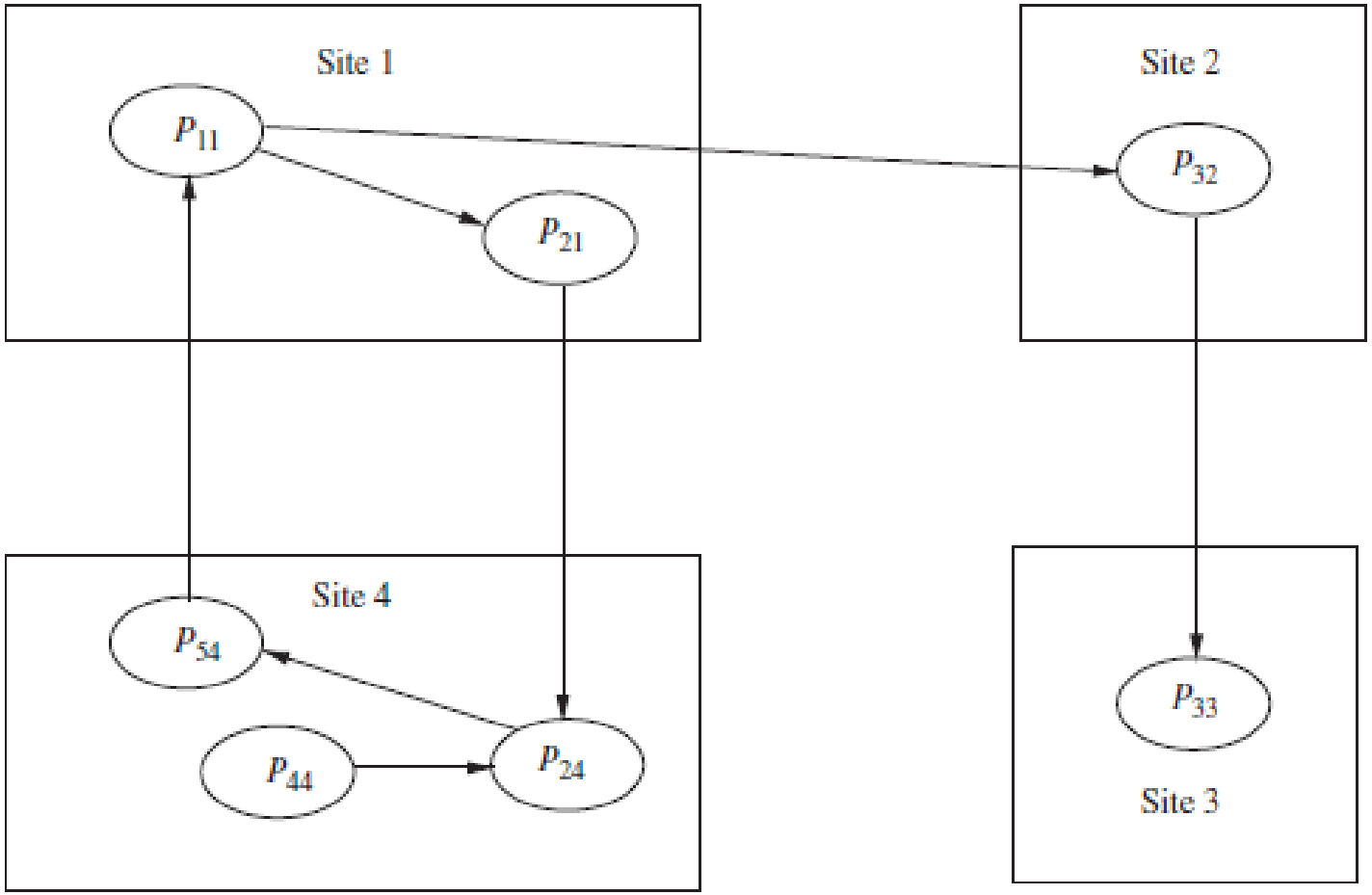
System Model

- N processors, N processes, each process runs on a processor
- systems have only reusable resources
- processes are allowed to make only exclusive access to resources
- only one copy/instance of each resource is present
- process can be in two states
  - running
  - blocked
- running state/active state:
  - process has all the needed resources
  - either is executing or is ready for execution
- blocked state - process is waiting to acquire some resource

Wait-For Graph (WFG)

- state of a distributed system can be modeled as a directed graph
  - ***wait-for graph (WFG)***
    - nodes are processes
    - a directed edge from node P_1 to node P_2 if
      - P_1 is blocked
      - P_1 is waiting for P_2 to release some resource
- a system is deadlocked if and only if there exists a directed cycle or knot in the WFG

Wait-for graph (WFG)



Wait-for graph (WFG)

- knot –
  - collection of vertices and edges such that every vertex in the knot has outgoing edges, and all outgoing edges from vertices in the knot terminate at other vertices in the knot
  - it is impossible to leave the knot while following any path along its directed edges
  - a vertex *v* is in a knot if for all *u* :: *u* is reachable from *v* : *v* is reachable from *u*

Wait-for graph (WFG)

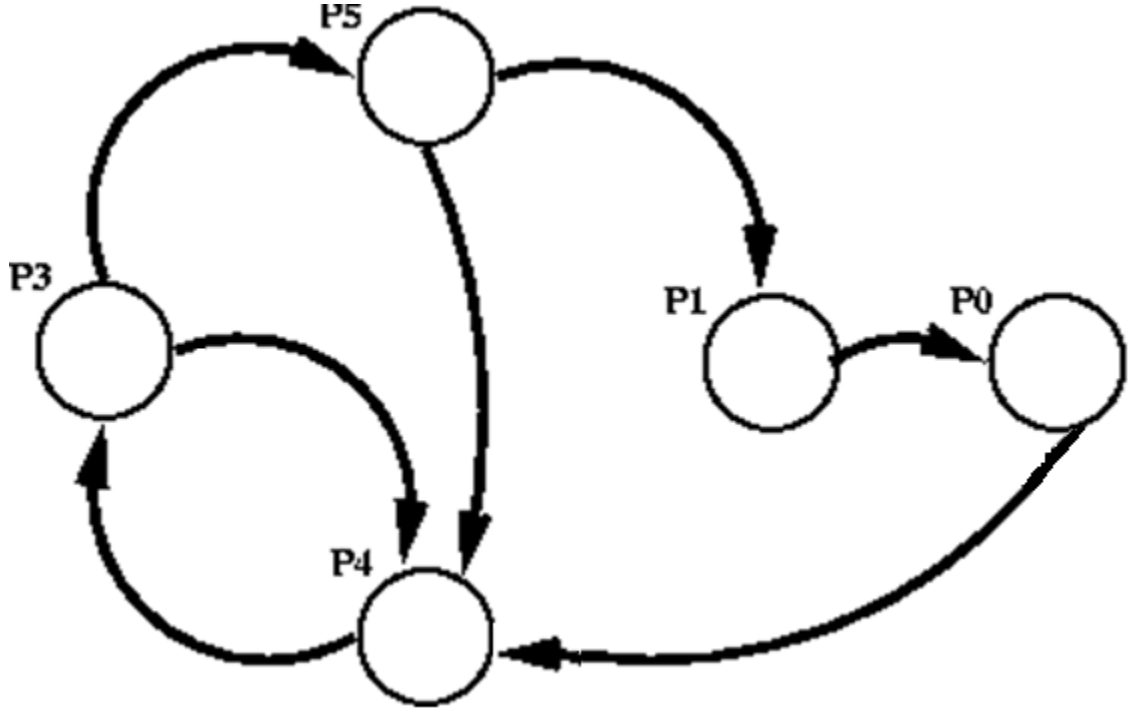
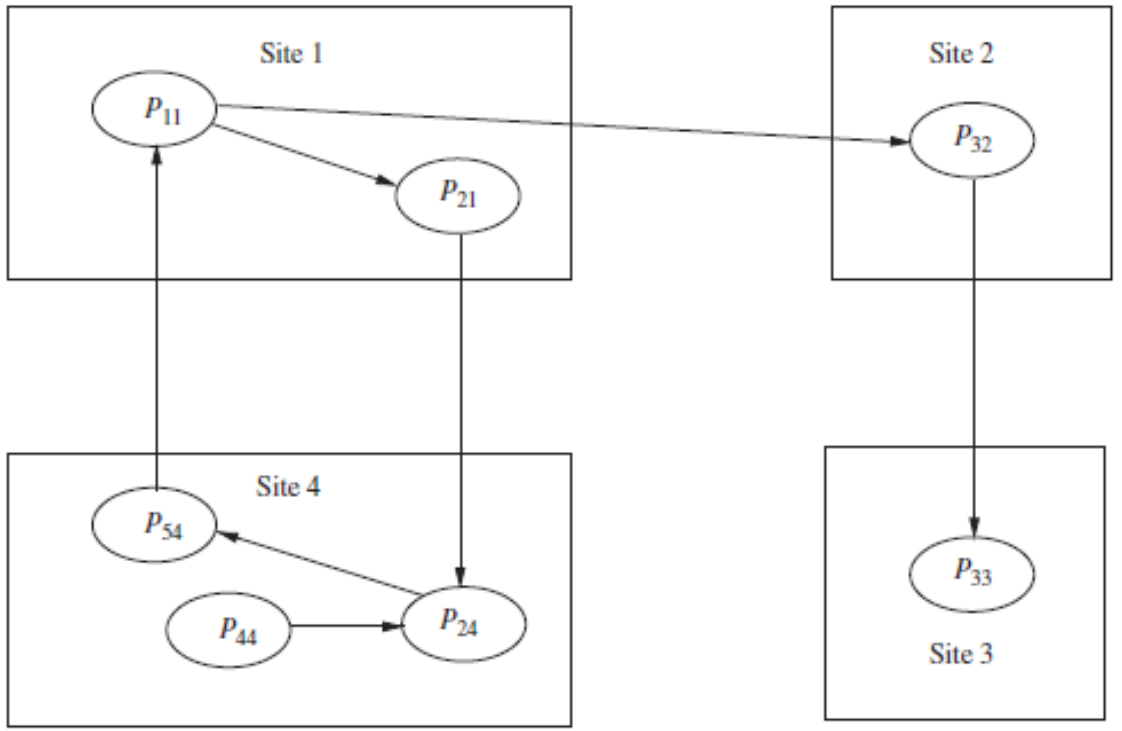


Image Source: https://www.cs.colostate.edu/~cs551/CourseNotes/Deadlock/WFGs.html

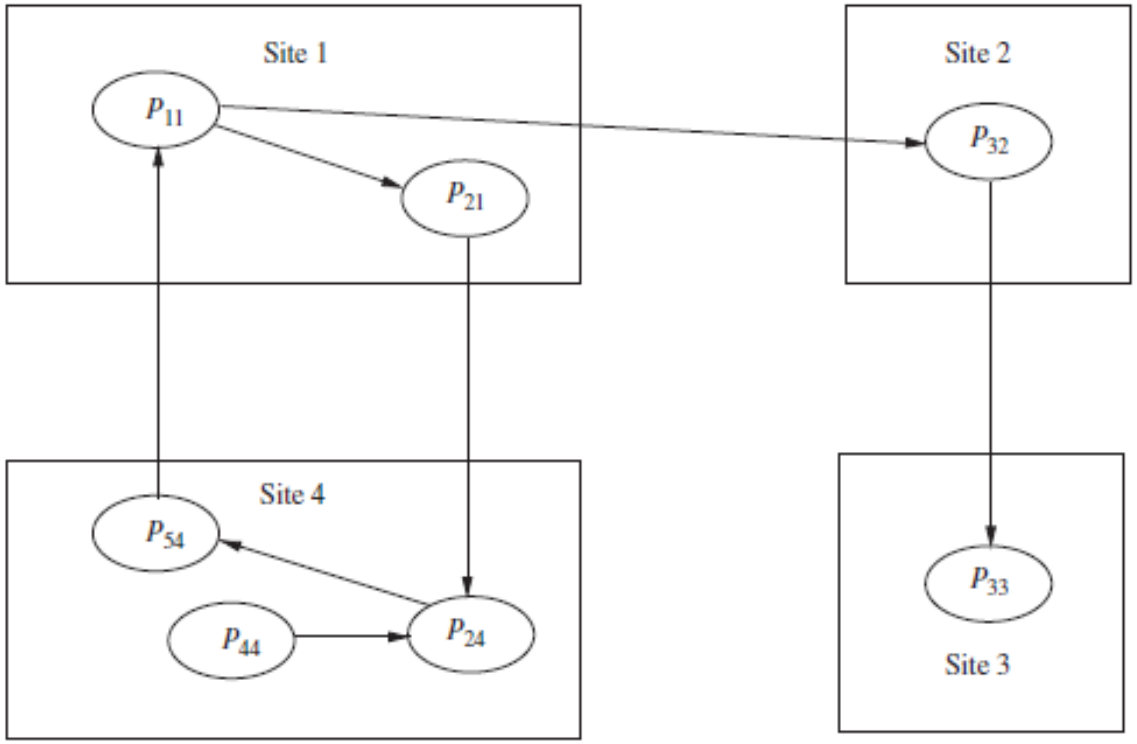
AND Model

- a process can request more than one resource simultaneously
- request is satisfied only after all the requested resources are granted to the process
- requested resources may exist at different sites
- out degree of a node in the WFG for AND model can be greater than 1
- presence of a cycle in the WFG indicates a deadlock
- each node of the WFG is an AND node



AND Model

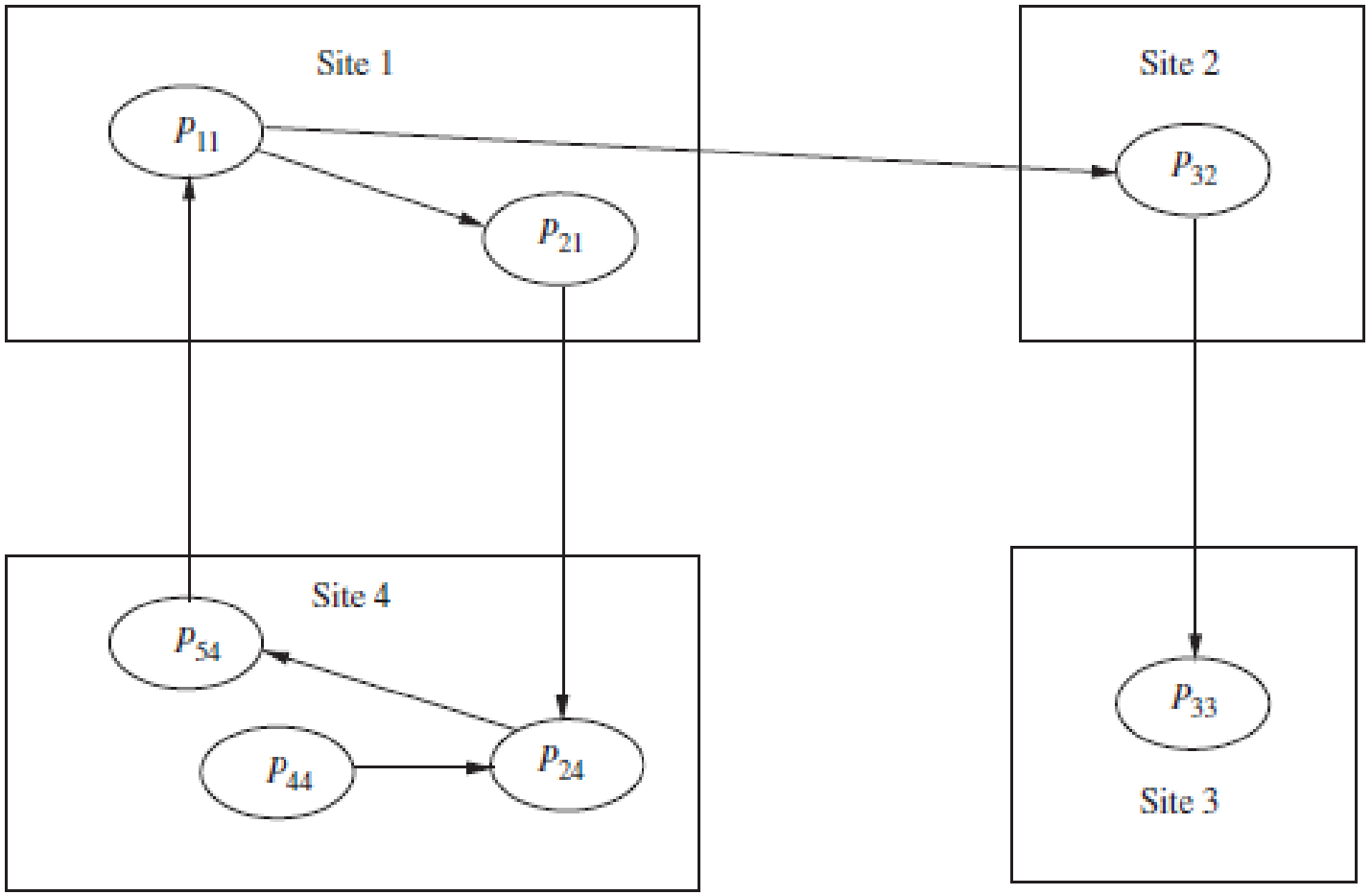
- if a cycle is detected in the WFG, it implies a deadlock but not vice versa
- a process may not be a part of a cycle, it can still be deadlocked



OR Model

- a process can make a request for multiple resources simultaneously
- request is satisfied if any one of the requested resources is granted
- requested resources may exist at different sites
- if all requests in the WFG are OR requests, then the nodes are called OR nodes
- **presence of a knot indicates a deadlock**
- with every blocked process, there is an associated set of processes called *dependent set*
- *a blocked process becomes active on receiving a grant message from any one of the processes in its dependent set*
- *a process is permanently blocked if it never receives a grant message from any of the processes in its dependent set*

OR Model



OR Model

- deadlock detection in the OR model is equivalent to finding knots in the graph
- note: there can be a deadlocked process that is not a part of a knot
- in an OR model, a blocked process P is deadlocked if it is either in a knot or it can only reach processes on a knot

Chandy–Misra–Haas Algorithm for the AND model

- uses a special message called **probe**
  - probe is a triplet **(i, j, k)**
  - denotes that
    - it belongs to a deadlock detection initiated for P_i (1^{st} element)
    - it is sent by the site of P_j (2^{nd} element)
    - it is sent to the site of P_k (3^{rd} element)
- probe message travels along the edges of the global WFG graph
- deadlock is detected when a probe message returns to the process that initiated it

Chandy–Misra–Haas Algorithm for the AND model

- P_j is said to be *dependent* on P_k if there exists a sequence of processes P_j, P_{i1}, P_{i2}, . . . , P_{im}, P_k such that
  - each process except P_k in the sequence is blocked
  - each process, except P_j, holds a resource for which the previous process in the sequence is waiting
- P_j is said to be *locally dependent* upon P_k if
  - P_j is dependent upon P_k
  - both the processes are on the same site

Chandy–Misra–Haas Algorithm for the AND model

Data structures

- each process P_i maintains a boolean array, $dependent_i$
- $dependent_i(j)$ is true if
  - P_i knows that P_j is dependent on it
- initially, $dependent_i(j)$ is false for all i and j

Chandy–Misra–Haas Algorithm for the AND model

if P_i is locally dependent on itself
then declare a deadlock
else for all P_j and P_k such that
(a) P_i is locally dependent upon P_j , and
(b) P_j is waiting on P_k, and
(c) P_j and P_k are on different sites,
send a probe (i, j, k) to the site of P_k

Chandy–Misra–Haas Algorithm for the AND model

On receipt of a probe (i, j, k), the site takes the following actions:

if

(d) P_k is blocked, and
(e) $dependent_k(i)$ is false, and
(f) P_k has not replied to all requests by P_j,

then

begin

$dependent_k(i)$ = true;

if k = i

then declare that P_i is deadlocked

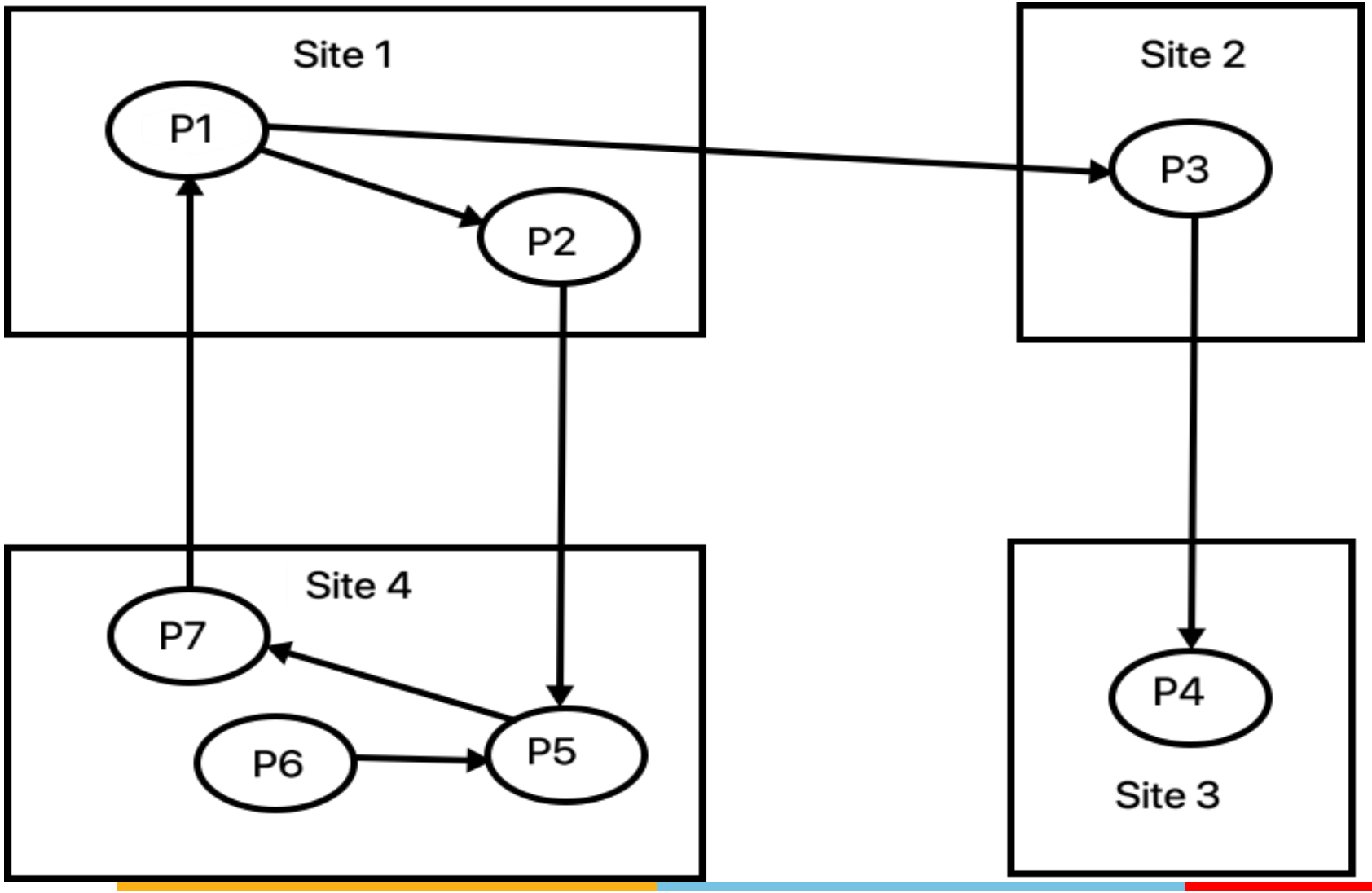
else for all P_m and P_n such that

(a') P_k is locally dependent upon P_m, and
(b') P_m is waiting on P_n, and
(c') P_m and P_n are on different sites,

send a probe (i, m, n) to the site of P_n

end

Chandy–Misra–Haas Algorithm for the AND model



Chandy–Misra–Haas Algorithm for the AND model

Performance analysis

- algorithm exchanges at most **m(n−1)/2** messages to detect a deadlock that involves **m** processes and spans over **n** sites

Chandy–Misra–Haas Algorithm for the OR model

- 2 types of messages are used:
  - query(i, j, k)
  - reply(i, j, k)
  - denote that they belong to a deadlock detection initiated by P_i and are being sent from P_j to P_k
- a blocked process initiates deadlock detection by sending query messages to all processes in its dependent set (i.e., processes from which it is waiting to receive a message)

Chandy–Misra–Haas Algorithm for the OR model

- local variable $num_k(i)$ = number of query messages sent
- P_k maintains a boolean variable $wait_k(i)$
- $wait_k(i)$ denotes that P_k has been continuously blocked since it received the last engaging query from P_i

Chandy–Misra–Haas Algorithm for the OR model

- first query message received by P_k for the deadlock detection initiated by P_i → ***engaging query***
- if an active process receives a query or reply message, it discards it
- when a blocked process P_k receives a query(i, j, k) message, it takes certain actions
  - if this is the engaging query from P_i, P_k propagates the query to all the processes in its dependent set and sets $num_k(i)$ = number of query messages sent
  - if this is not engaging query, P_k returns a reply message if P_k has been continuously blocked since it received the corresponding engaging query
  - if P_k is not blocked, it discards the query

Chandy–Misra–Haas Algorithm for the OR model

Initiate a deadlock detection for a blocked process P_i:

- send query(i, i, j) to all processes P_j in the dependent set DS_i of P_i
- $num_i(i) = |DS_i|$
- $wait_i(i)$ = true

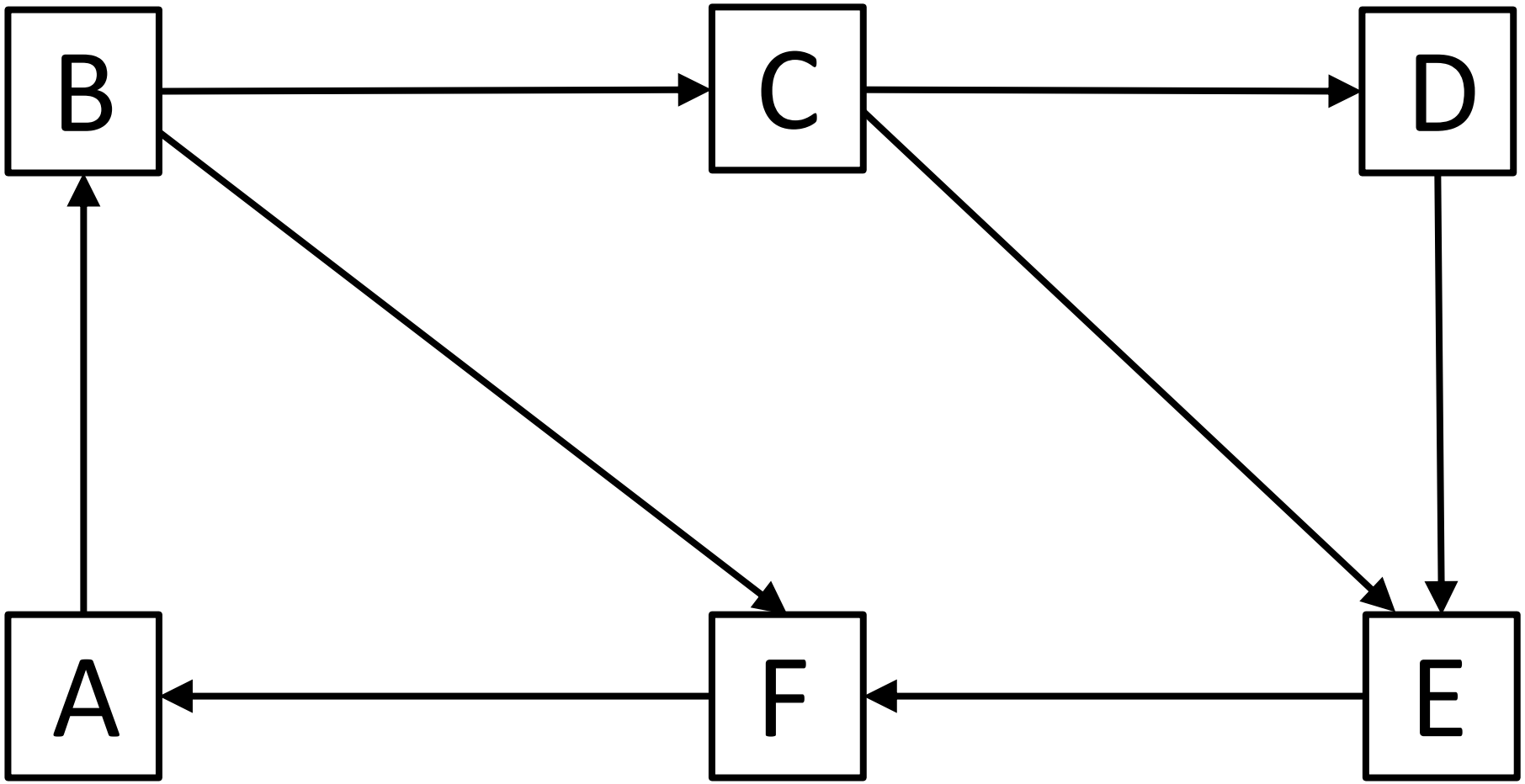
When a blocked process P_k receives a query(i, j, k):

- ***if*** this is the **engaging query** for process P_i then
  - send query(i, k, m) to all P_m in its dependent set DS_k
  - $num_k(i) = |DS_k|$
  - $wait_k(i)$ = true
- ***else*** if $wait_k(i)$ = true then
  - send a reply(i, k, j) to P_j

Chandy–Misra–Haas Algorithm for the OR model

When a process P_k receives a reply(i, j, k):

- ***if*** $wait_k(i)$ = true then
  - $num_k(i) = num_k(i) - 1$
- ***if*** $num_k(i) = 0$ then
  - ***if*** i = k then **declare a deadlock**
  - ***else*** send reply(i, k, m) to the process P_m which sent the engaging query


B
C
D
A
F
E

Chandy–Misra–Haas Algorithm for the OR model

Performance analysis

- for every deadlock detection, the algorithm exchanges **e** query messages and **e** reply messages, where **e = n(n – 1)**
- e is the number of edges, n is the number of processes

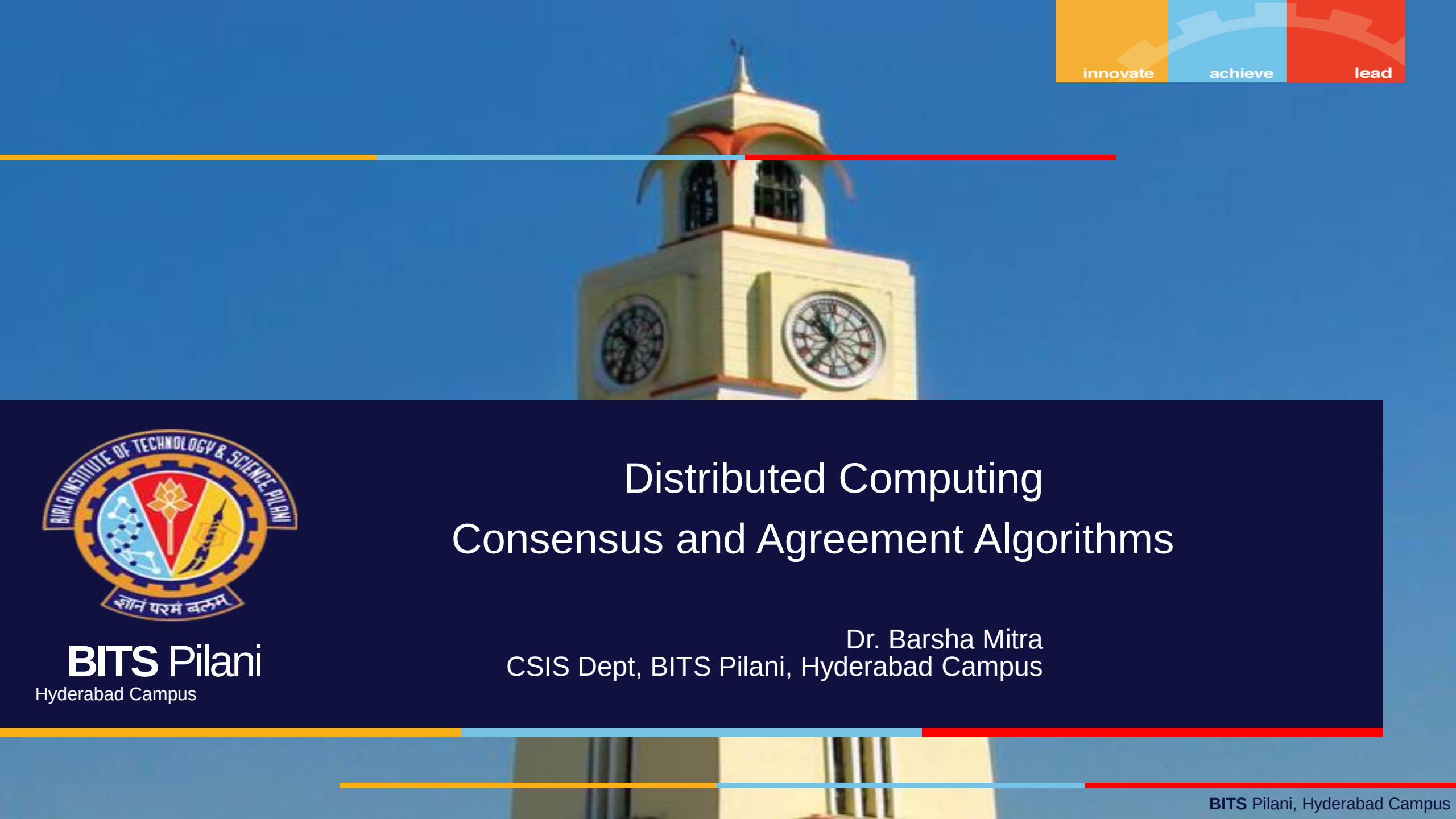
Deadlock Resolution

- deadlock resolution involves breaking existing wait-for dependencies between processes
- rolling back one or more deadlocked processes
- assigning resources of rolled back processes to blocked processes
- blocked processes resume execution
- when a wait-for dependency is broken
  - the corresponding information should be immediately cleaned from the system
- otherwise may result in detection of phantom deadlocks

References

- Ajay D. Kshemkalyani, and Mukesh Singhal, Chapter 10, “Distributed Computing: Principles, Algorithms, and Systems”, Cambridge University Press, 2008.
- http://greenteapress.com/complexity/html/thinkcomplexity014.html
- ube.ege.edu.tr/~cinsdiki/2006...8.../Chandy-Misra-Haas-DistDeadlockDetection.ppt

Thank You



innovate achieve lead

Distributed Computing
Consensus and Agreement Algorithms

Dr. Barsha Mitra
CSIS Dept, BITS Pilani, Hyderabad Campus

BITS Pilani
Hyderabad Campus

Introduction

- agreement among processes in a distributed system
- processes need to exchange information to negotiate with one another and eventually reach a common understanding or agreement, before taking actions

Assumptions for Agreement Algorithms

Failure models

- among the **n** processes in the system, at most **f** processes can be faulty
- faulty process can behave in any manner allowed by the failure model assumed
- various processor failure models – From recorded lecture

Assumptions for Agreement Algorithms

Synchronous/Asynchronous communication

- if a failure-prone process chooses to send a message to process P_i but fails, then P_i cannot detect the non-arrival of the message in an asynchronous system
- scenario is indistinguishable from the scenario in which the message takes a very long time to travel
- impossible to reach a consensus in asynchronous systems

Assumptions for Agreement Algorithms

Synchronous/Asynchronous communication (contd..)

- **synchronous system** –
  - a message that has not been sent can be recognized by the intended recipient, at the end of the round
  - intended recipient can deal with the non-arrival of the expected message by assuming the arrival of a message containing some default data, and then proceeding with the next round of the algorithm

Network connectivity

- system has full connectivity, i.e., each process can communicate with any other by direct message passing

Assumptions for Agreement Algorithms

Sender identification

- a process receiving a message always knows identity of sender process
- also true for malicious senders

Channel reliability

- channels are reliable
- only the processes may fail (under one of various failure models)

Agreement variable

- agreement variable
  - may be boolean or multivalued
  - need not be an integer

Assumptions for Agreement Algorithms

Non-authenticated messages

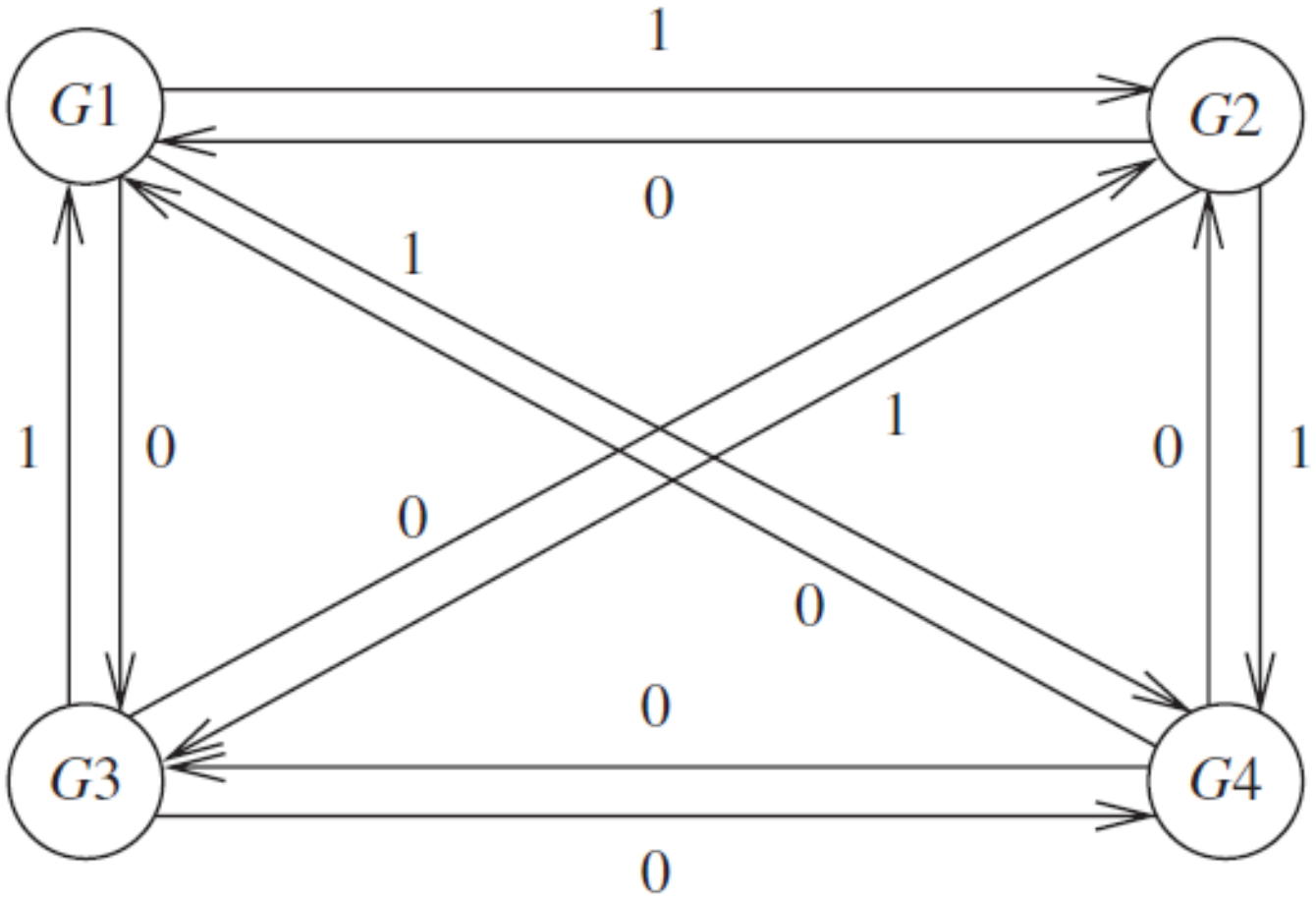
- with unauthenticated messages, when a faulty process relays a message to other processes
  - it can forge the message and claim that it was received from another process
  - it can also tamper with the contents of a received message before relaying it
- receiver cannot verify its authenticity
- unauthenticated message is also called oral message or unsigned message

Assumptions for Agreement Algorithms

Authenticated messages

- using authentication via techniques such as digital signatures
- if some process forges a message or tampers with the contents of a received message before relaying it, the recipient can detect the forgery or tampering

Byzantine Agreement Problem



Byzantine Agreement Problem

- requires a designated process, called the **source process**, with an initial value, to reach agreement with the other processes about its initial value, subject to the following conditions:
  - **Agreement:** all non-faulty processes must agree on the same value
  - **Validity:** if the source process is non-faulty, then the agreed upon value by all the non-faulty processes must be the same as the initial value of the source
  - **Termination:** each non-faulty process must eventually decide on a value

Byzantine Agreement Problem

- if the source process is faulty, then the correct processes can agree upon any default value
- irrelevant what the faulty processes agree upon – or whether they terminate and agree upon anything at all

Consensus Problem

- **Initiated by all processes**
- **Agreement:** all non-faulty processes must agree on the same (single) value
- **Validity:**
  - if all the non-faulty processes have the same initial value, then the agreed upon value by all the non-faulty processes must be that same value
  - if non-faulty processes broadcast different initial values, then these processes should decide upon a common value
- **Termination:** each non-faulty process must eventually decide on a value

Interactive Consistency Problem

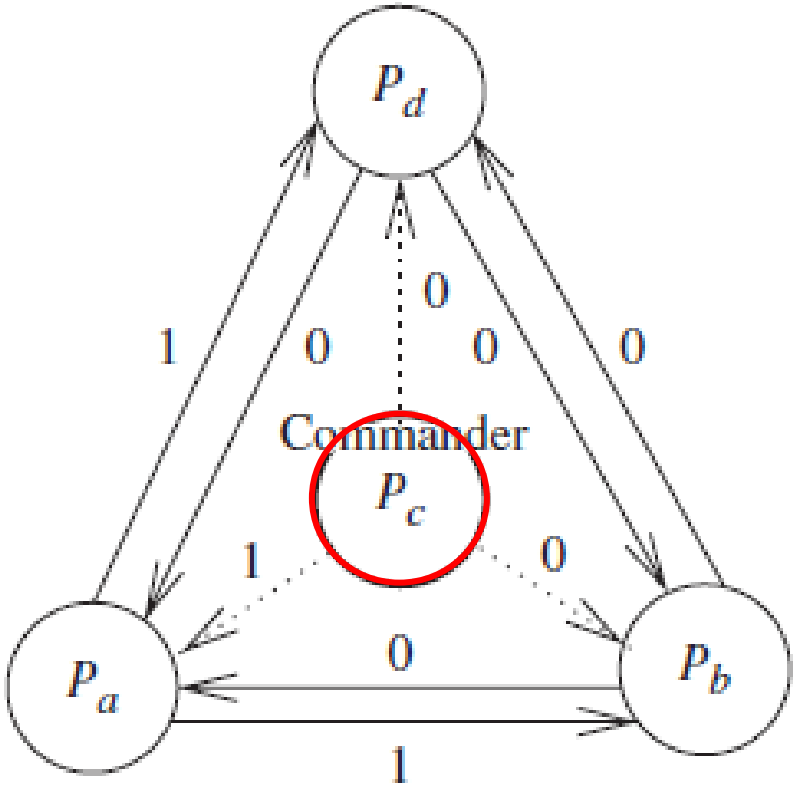
- **Initiated by all processes**
- **Agreement:** all non-faulty processes must agree on the same array of values A[v_1…v_n]
- **Validity:**
  - if process P_i is non-faulty and its initial value is v_i, then all non-faulty processes agree on v_i as the i^{th} element of the array A
  - if P_j is faulty, then the non-faulty processes can agree on any value for A[j]
- **Termination:** each non-faulty process must eventually decide on the array A

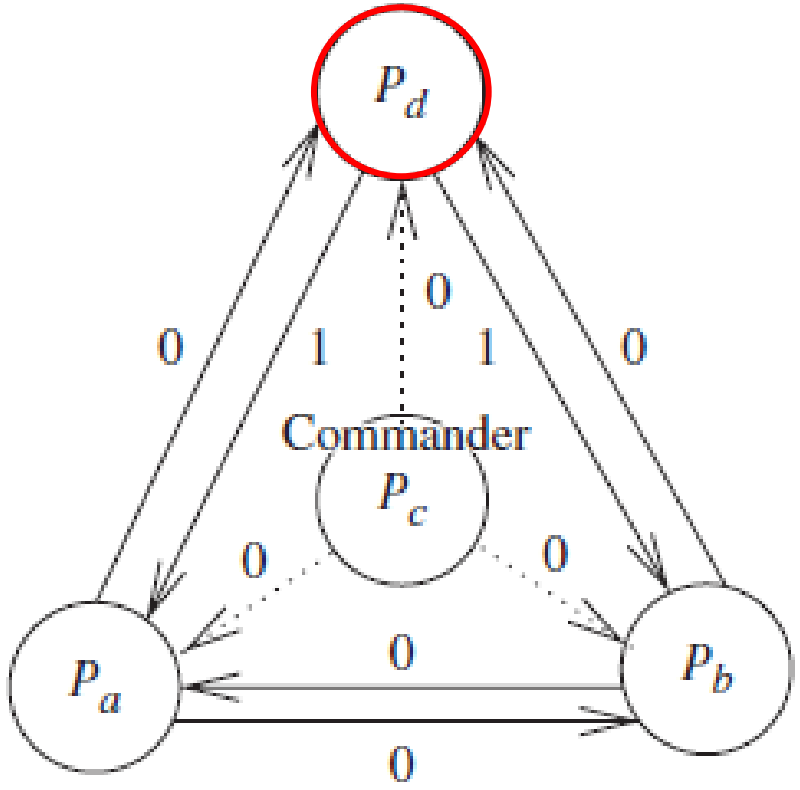
Upper Bound on Byzantine Processes

in a system of **n** processes, the Byzantine agreement problem (and the other variants of the agreement problem) can be solved in a synchronous system only if the number of Byzantine processes **f** is such that

$$f \leq \lfloor (n-1)/3 \rfloor$$

Byzantine Agreement Tree Algorithm: Synchronous System





Byzantine Agreement Tree Algorithm: Recursive Formulation

- Oral Message algorithm, OM(f) consists of f+1 "phases/rounds"
- Algorithm OM(0) is the "base case" (no faults)

  1)Commander sends value to every lieutenant

  2)Each lieutenant uses value received from commander, or a default value if no value was received
- Recursive algorithm OM(f) handles up to f faults

  1)Commander sends value to every lieutenant

  2)For each lieutenant *i*, let v_i be the value *i* received from commander, or a default value if no value was received. Lieutenant *i* acts as commander and runs Alg OM(f -1) to send v_i to each of the *n-2* other lieutenants

  3)For each *i*, and each *j* not equal to *i,* let v_j be the value lieutenant *i* received from lieutenant *j* in step (2) (using Alg OM(f-1)), or else a default value if no such value was received. Lieutenant *i* uses the value *majority*$(v_1, \ldots, v_{n-1})$ *to compute the agreed upon value.*

Byzantine Agreement Tree Algorithm: Recursive Formulation

Byzantine Agreement Tree Algorithm: Recursive Formulation

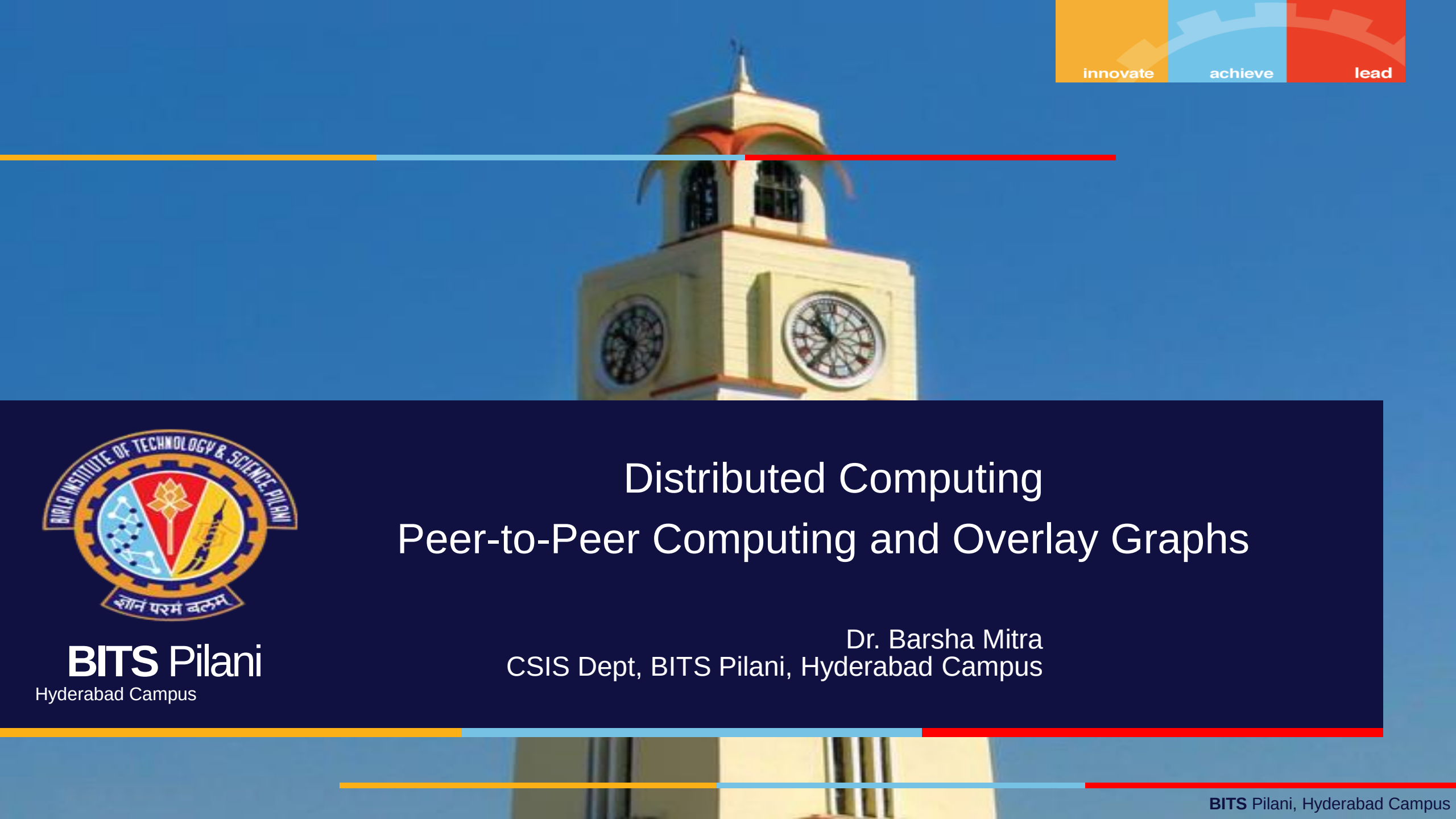
Round number	A message has already visited	Aims to tolerate these many failures	Each message gets sent to	Total number of messages in round
1	1	f	n-1	n-1
2	2	f-1	n-2	(n-1)(n-2)
…	…	…	…	..
x	x	(f+1)-x	n-x	(n-1)(n-2)…(n-x)
x+1	x+1	(f+1)-x-1	n-x-1	(n-1)(n-2)…(n-x-1)
…	…	…	…	…
f+1	f+1	0	n-f-1	(n-1)(n-2)…(n-f-1)

exponential number of messages $O(n^f)$ used

Reference

- Ajay D. Kshemkalyani, and Mukesh Singhal, Chapter 14, “Distributed Computing: Principles, Algorithms, and Systems”, Cambridge University Press, 2008.

Thank You



innovate achieve lead

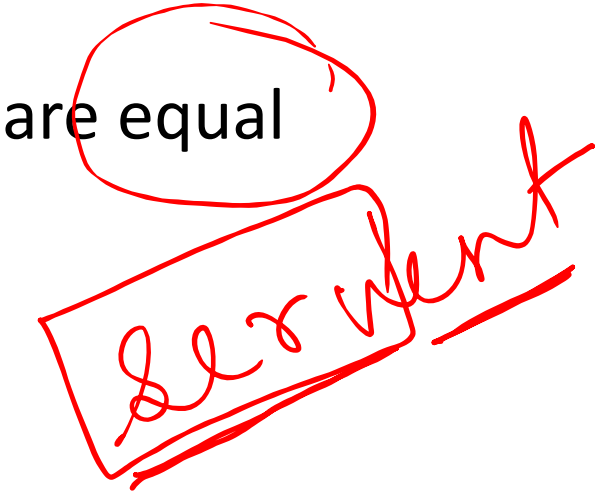
BITS Pilani
Hyderabad Campus

Distributed Computing
Peer-to-Peer Computing and Overlay Graphs

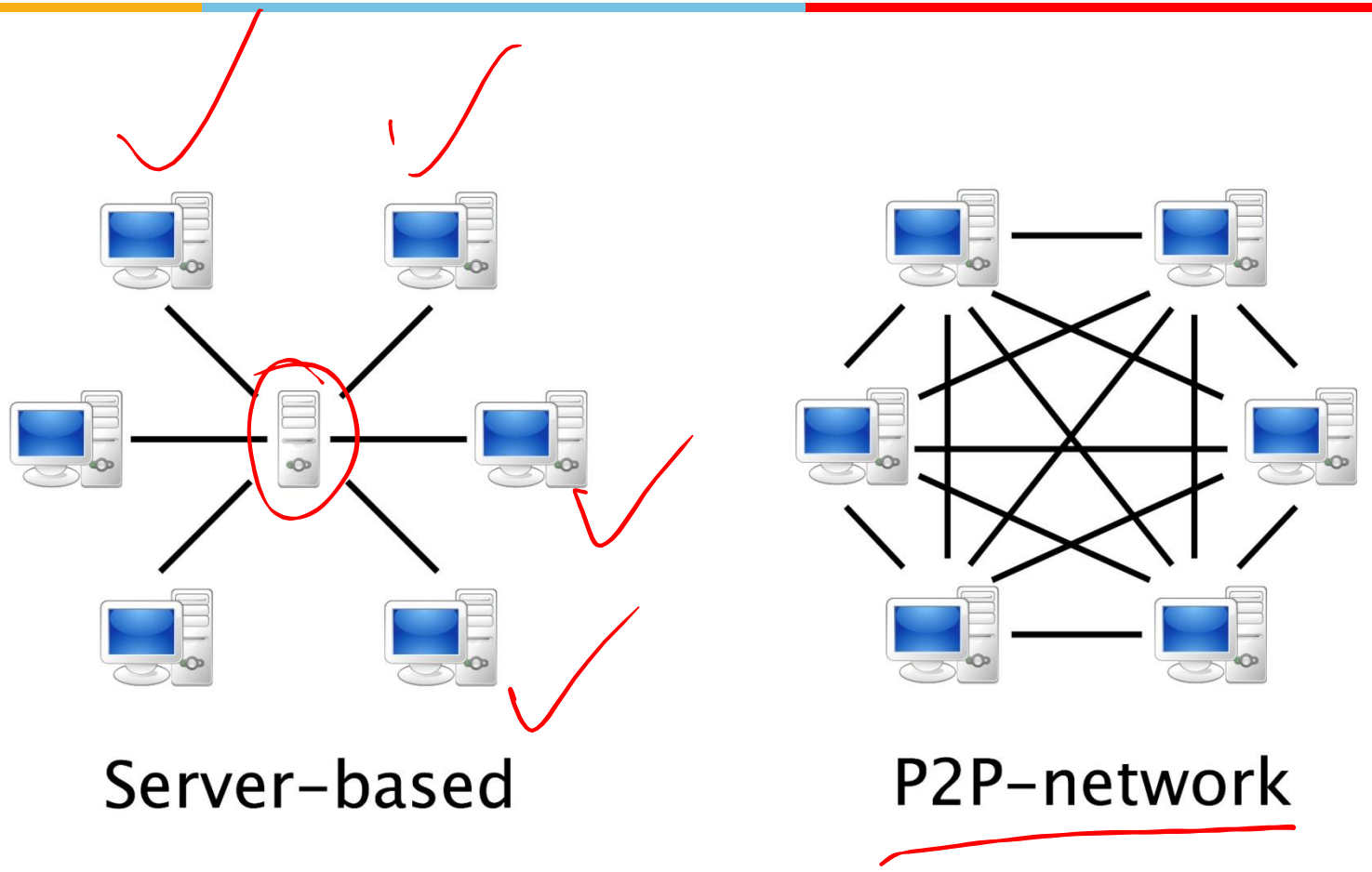
Dr. Barsha Mitra
CSIS Dept, BITS Pilani, Hyderabad Campus

Introduction

- allows for flexibly sharing resources (e.g., files and multimedia documents) stored across network-wide computers
- any node in a P2P network can act as a server to others and also as a client
- communication and exchange of information is performed directly between the participating peers
- relationships between the nodes in the network are equal



Introduction



source: http://www.terndrup.net/2015/10/27/Building-a-P2P-Peer-Client-with-Node-js/

Introduction

- well known P2P networks that allow P2P file-sharing
  - Napster
  - Gnutella
  - Freenet
  - Pastry
  - Chord
  - CAN

Introduction

- P2P networks
  - allow the location of arbitrary data objects
  - impose a low cost for scalability, and for entry into and exit from the network
- ongoing entry and exit of various nodes and dynamic insertion and deletion of objects is termed as ***churn***
- impact of churn should be as transparent as possible

Data Indexing and Overlays

- data in a P2P network is identified by using indexing
- indexing mechanisms can be classified as:
  - centralized
  - local
  - distributed

Data Indexing and Overlays

Centralized Indexing:

- use of one or a few central servers to store references (indexes) to the data on many peers
- eg. Napster

Local indexing:

- requires each peer to index only the local data objects
- remote objects need to be searched for
- used in unstructured overlays
- Gnutella uses local indexing

Data Indexing and Overlays

Distributed Indexing:

- involves the indexes to the objects at various peers being scattered across other peers throughout the P2P network
- distributed indexing is the most challenging of the indexing schemes
- many novel mechanisms have been proposed, most notably the distributed hash table (DHT)

Structured Overlays

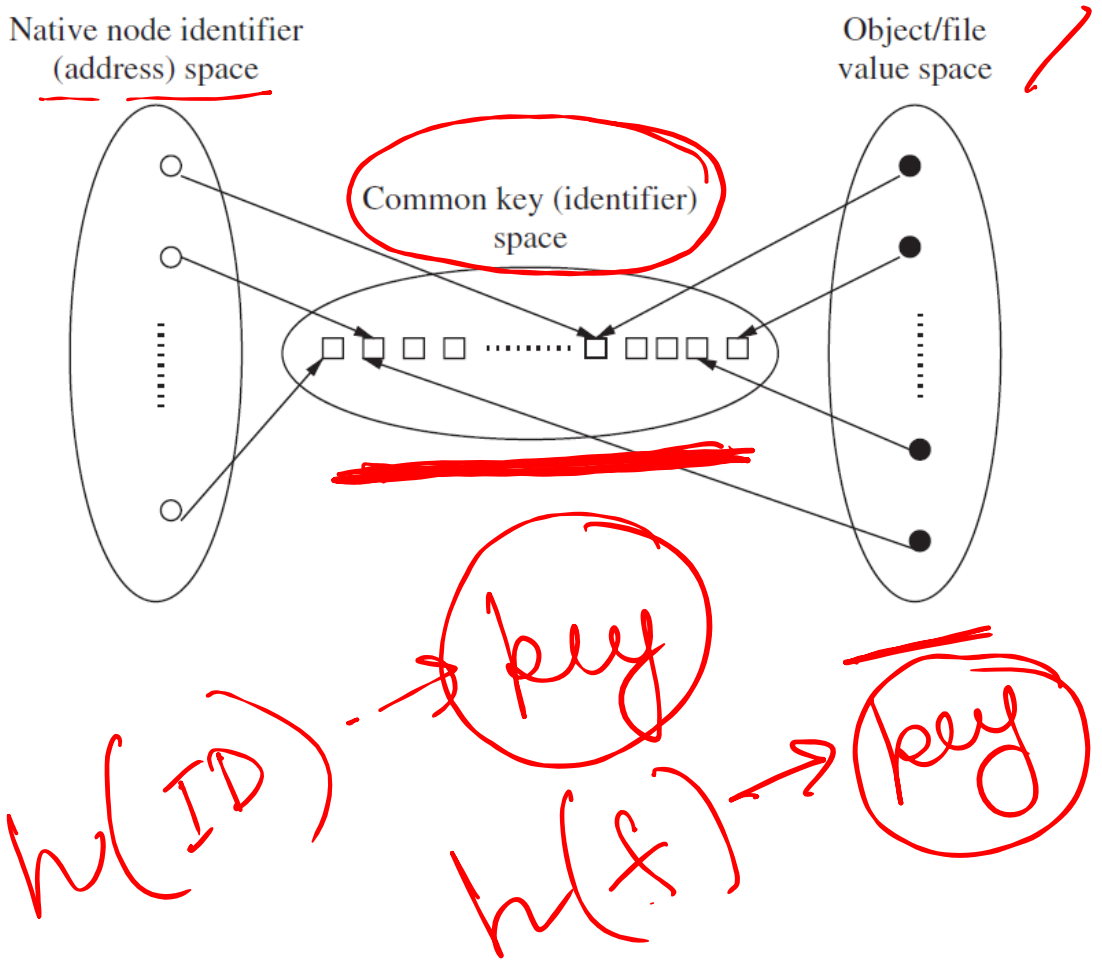
- P2P network topology has a definite structure
- placement of files or data in this network is highly deterministic as per some algorithmic mapping
- use a hash table interface for the mapping
- hash function maps keys to values
- allows fast search for the location of a file
- disadvantage of such mapping:
  - arbitrary queries such as range queries, attribute queries and exact keyword queries cannot be handled directly

Unstructured Overlays

- P2P network topology does not have any particular controlled structure
- no control over where files/data is placed
- local indexing is used
- file placement is not governed by the topology
- search for a file may have high message overhead and high delays
- queries may be unsuccessful even if the queried object exists
- complex queries are supported because the search criteria can be arbitrary

Chord distributed hash table: Overview

- uses a flat key space to associate the mapping between network nodes and data objects/files/values
- node address and data object/file/value is mapped to a logical identifier in the common key space using a hash function
- equal distribution of keys among nodes

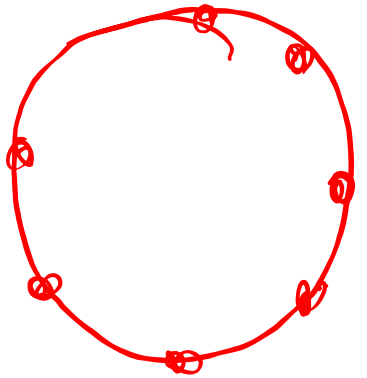


Chord distributed hash table: Overview

- supports a single operation, lookup(x)
- lookup(x) maps a given key x to a network node
- Chord stores a file/object/value at the node to which the file/object/value's key maps

Chord distributed hash table: Overview

- node's IP address is hashed to an m-bit identifier that serves as the node identifier in the common key (identifier) space
- file/data key is hashed to an m-bit identifier that serves as the key identifier
- identifier space is ordered on the logical ring modulo 2^m
- a key k gets assigned to the first node such that its node identifier equals or follows the key identifier of k in the common identifier space
- node is the successor of k, denoted succ(k)

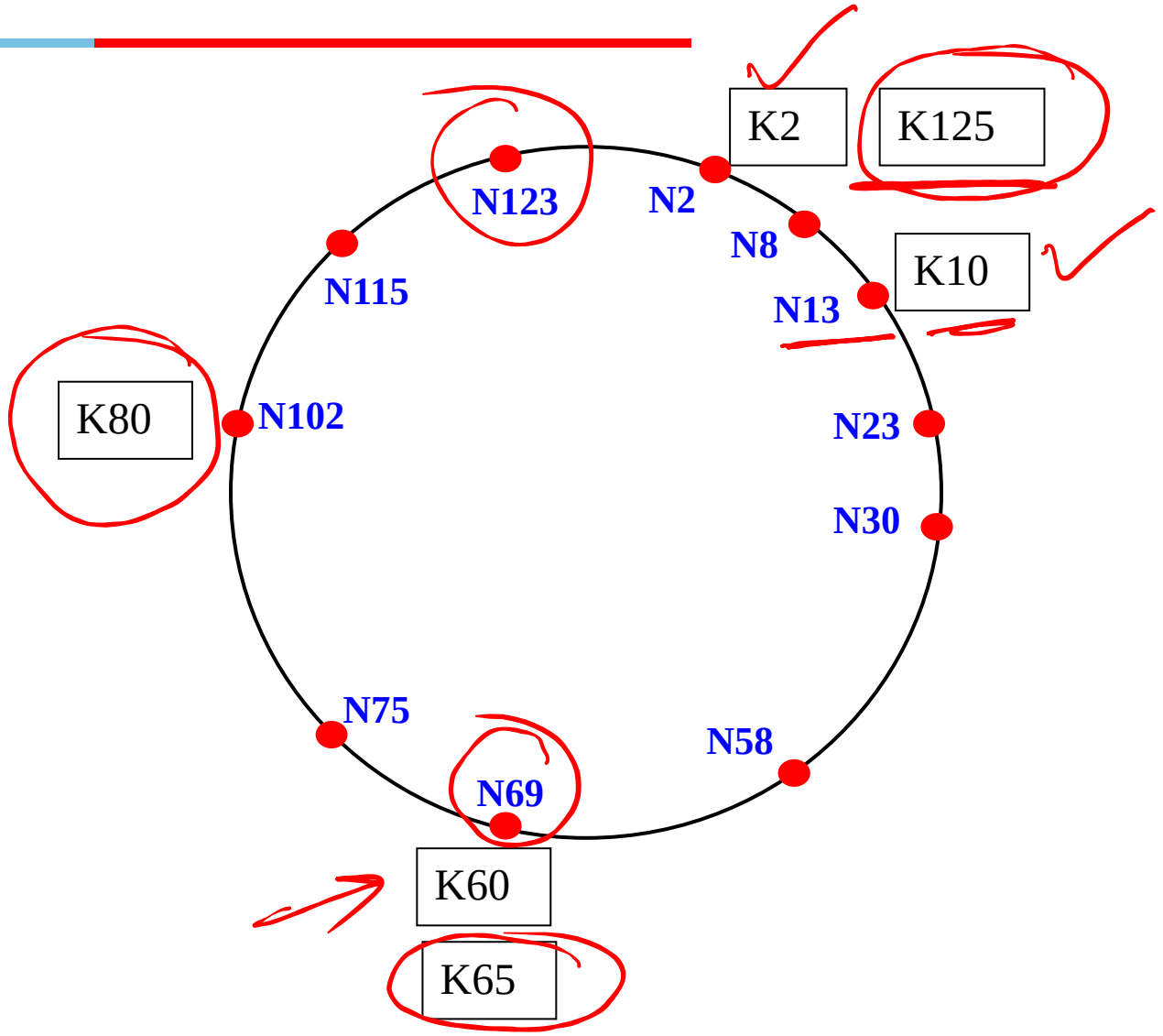


Chord distributed hash table: Overview

- m = 7
- succ(2) = 2
- succ(10) = 13
- succ(60) = 69
- succ(65) = 69
- succ(80) = 102
- succ(125) = 2
- succ(148) = 23

Convention:

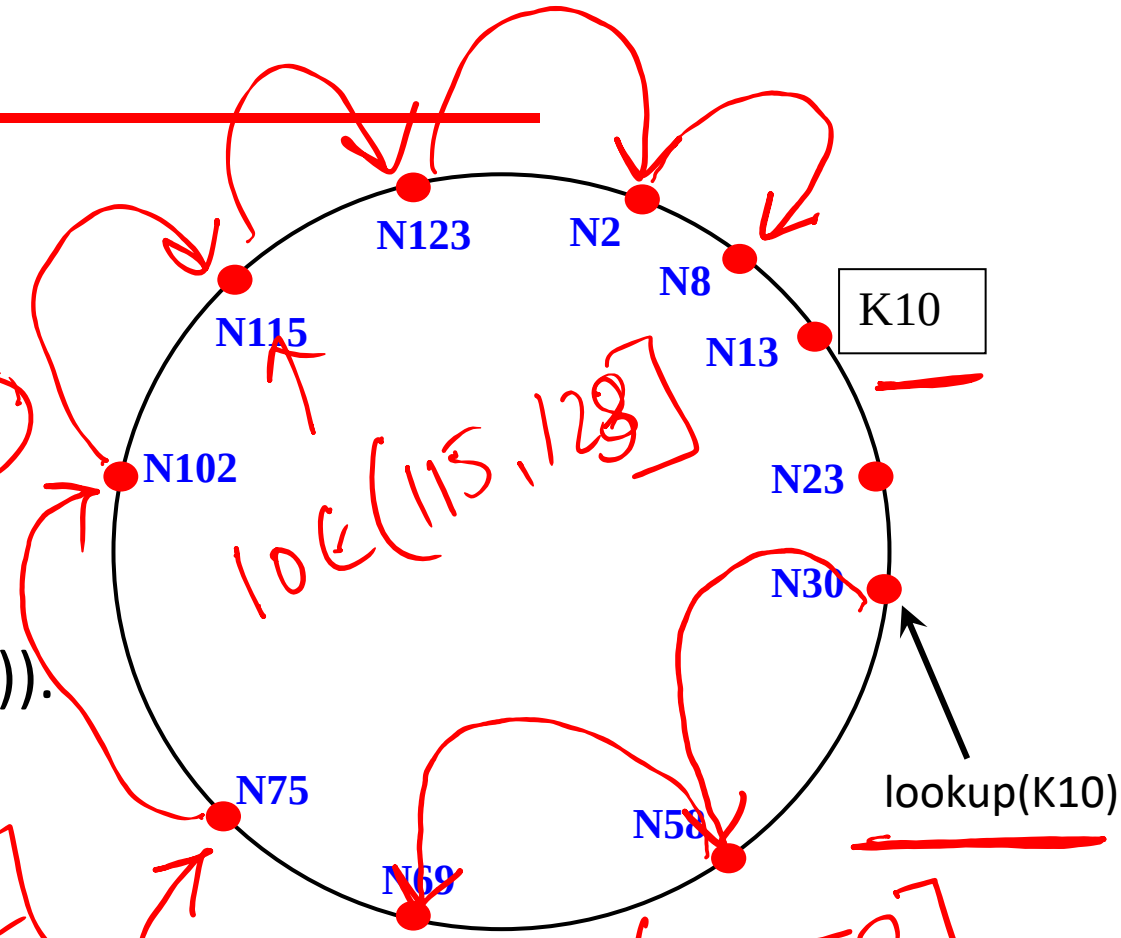
- (x, y] → left-open right-closed segment of the Chord ring



Simple Lookup

(variable)
integer: successor ← initial value;

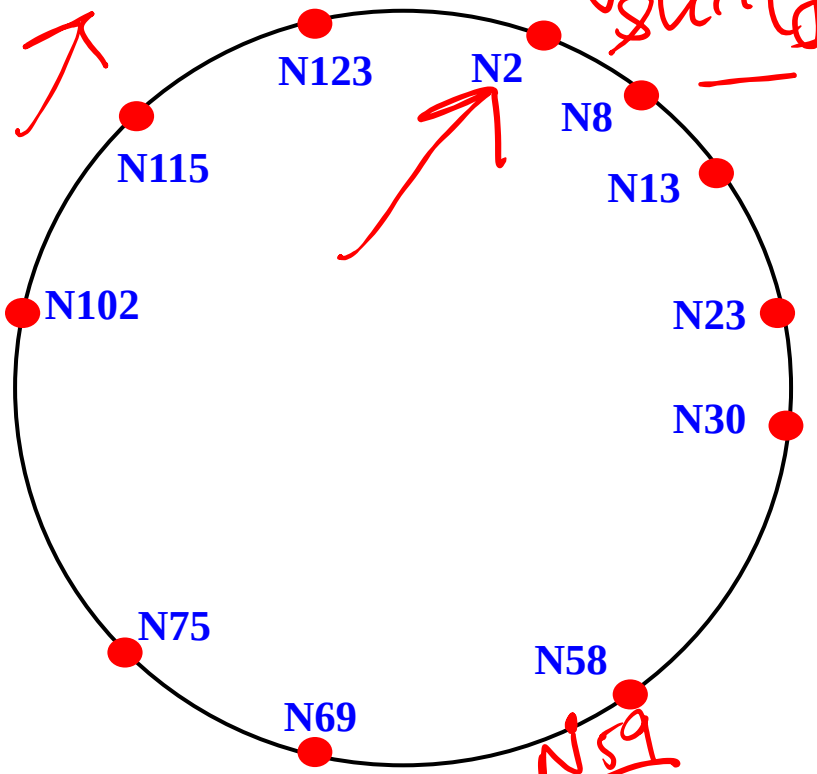
(1) i.Locate_Successor(*key*), where *key* is not at i:
(1a) **if** *key* ∈ (i, successor] **then**
(1b) **return** (successor)
(1c) **else return** (successor.Locate_Successor(*key*)).



Scalable Lookup

- each node i maintains a routing table, called *finger table*
- x^{th} entry ($1 \le x \le m$) is the node identifier of the node succ($i+2^{x-1}$)
- size of the finger table is bounded by m entries

N8	N58	N102
8 + 1 = 9 – N13	58 + 1 = 59 – N69	102 + 1 = 103 – N115
8 + 2 = 10 – N13	58 + 2 = 60 – N69	102 + 2 = 104 – N115
8 + 4 = 12 – N13	58 + 4 = 62 – N69	102 + 4 = 106 – N115
8 + 8 = 16 – N23	58 + 8 = 66 – N69	102 + 8 = 110 – N115
8 + 16 = 24 – N30	58 + 16 = 74 – N75	102 + 16 = 118 – N123
8 + 32 = 40 – N58	58 + 32 = 90 – N102	102 + 32 = 134 – N8
8 + 64 = 72 – N75	58 + 64 = 122 – N123	102 + 64 = 166 – N58



Scalable Lookup

- search is highly scalable
- for a query on key *key* at node i, if *key* lies between i and its successor, then *key* would reside at the successor and the successor's address is returned
- else the finger table is searched

(variables)

integer: successor ← initial value;
integer: predecessor ← initial value;
integer: finger[1…m];

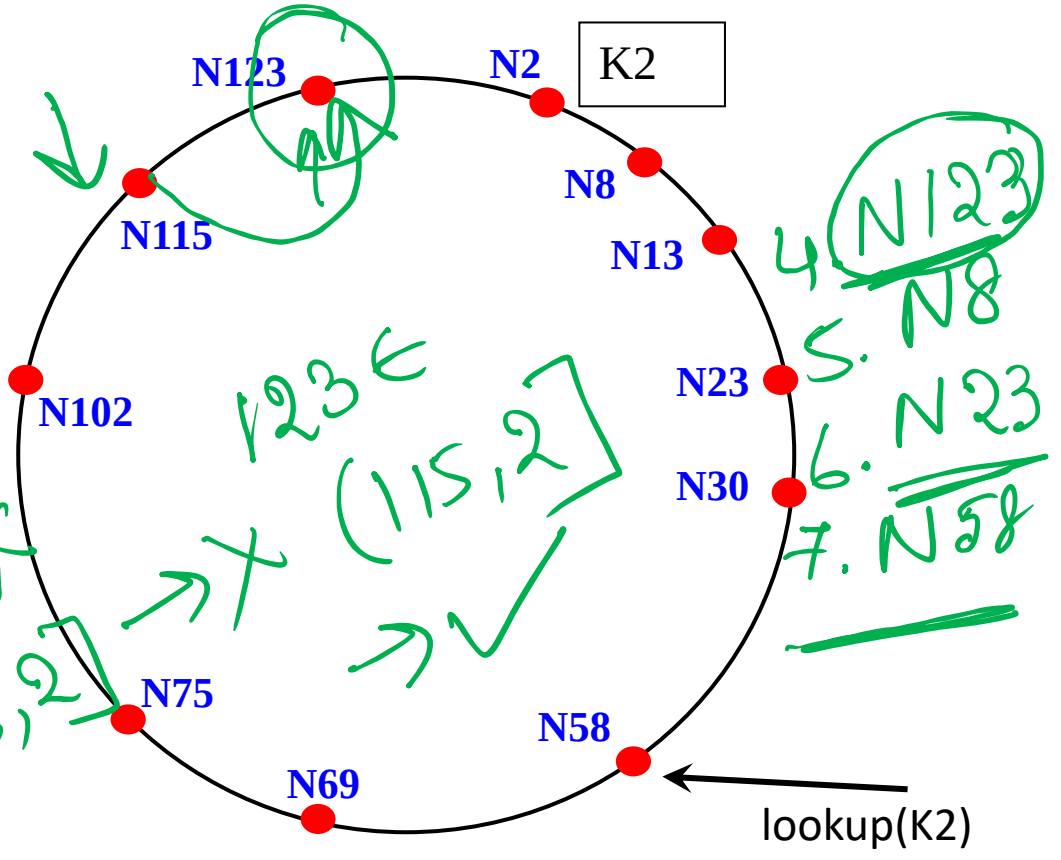
key ∈ (i, succ]

Scalable Lookup

N8	N58	N102
8 + 1 = 9 – N13	58 + 1 = 59 – N69	102 + 1 = 103 – N115
8 + 2 = 10 – N13	58 + 2 = 60 – N69	102 + 2 = 104 – N115
8 + 4 = 12 – N13	58 + 4 = 62 – N69	102 + 4 = 106 – N115
8 + 8 = 16 – N23	58 + 8 = 66 – N69	102 + 8 = 110 – N115
8 + 16 = 24 – N30	58 + 16 = 74 – N75	102 + 16 = 118 – N123
8 + 32 = 40 – N58	58 + 32 = 90 – N102	102 + 32 = 134 – N8
8 + 64 = 72 – N75	58 + 64 = 122 – N123	102 + 64 = 166 – N58

(1) i.Locate_Successor(*key*), where *key* is not at i:
(1a) **if** *key* ∈ (i, successor] **then**
(1b) **return** (successor)
(1c) **else**
(1d) j ← Closest_Preceding_Node(*key*);
(1e) **return** (j.Locate_Successor(*key*))

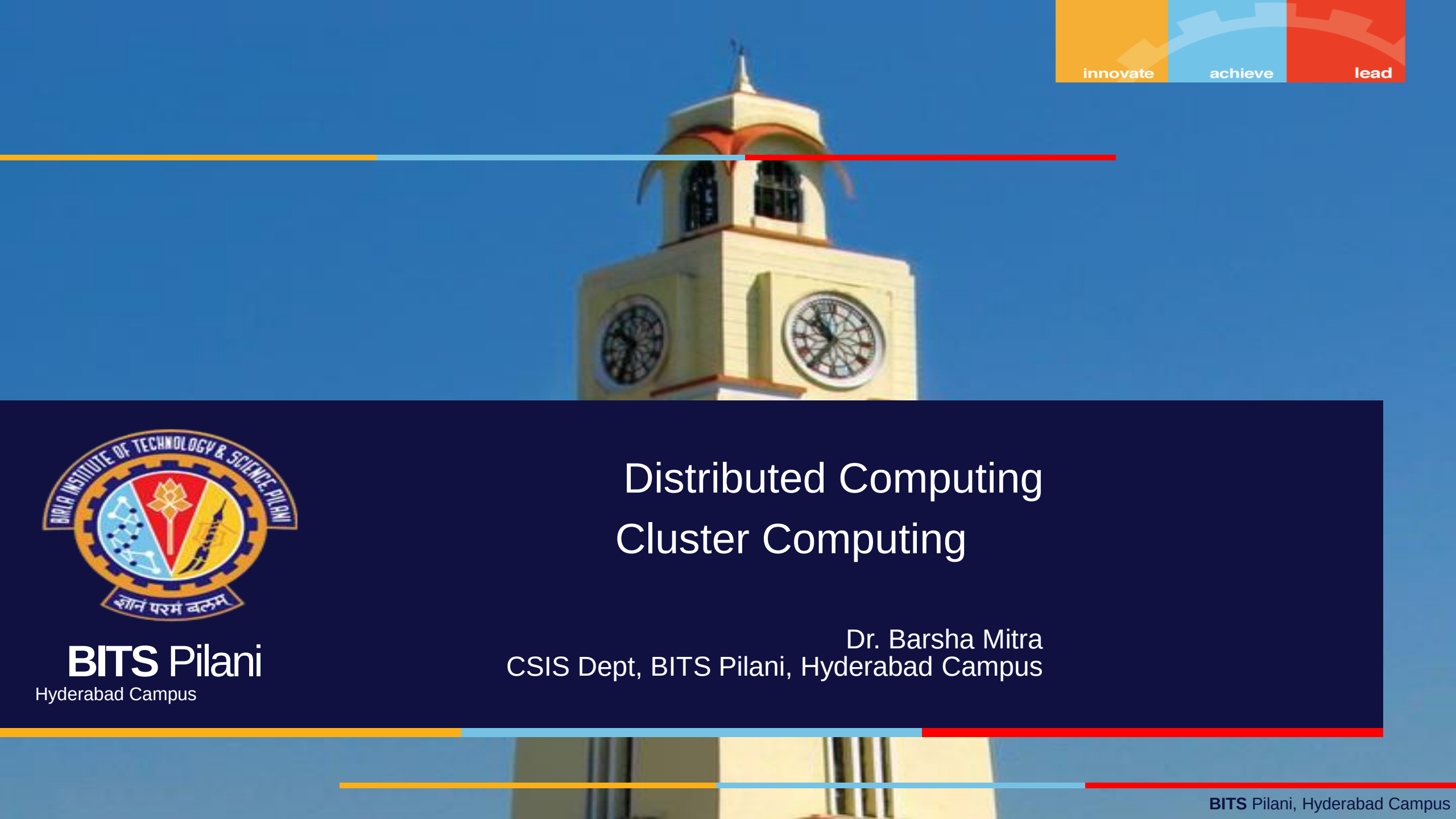
(2) i.Closest_Preceding_Node(*key*), where *key* ≠ i:
(2a) **for** count = m **down to** 1 **do**
(2b) **if** finger[count] ∈ (i, *key*] **then**
(2c) **break()**;
(2d) **return** (finger[count])



References

- Ajay D. Kshemkalyani, and Mukesh Singhal, Chapter 18, "Distributed Computing: Principles, Algorithms, and Systems", Cambridge University Press, 2008.

Thank You



innovate achieve lead

Distributed Computing
Cluster Computing

Dr. Barsha Mitra
CSIS Dept, BITS Pilani, Hyderabad Campus

BITS Pilani
Hyderabad Campus

Clustering for Massive Parallelism

- computer cluster
  - consists of a collection of interconnected stand-alone/complete computers
  - cooperatively work together as a single, integrated computing resource
  - explores parallelism at job level

Clustering for Massive Parallelism

- benefits of computer clusters
  - scalable performance
  - HA
  - fault tolerance
  - modular growth
  - use of commodity components

Design Objectives of Computer Clusters

Scalability

- scalability could be limited by a number of factors, such as the multicore chip technology, cluster topology, packaging method, power consumption, and cooling scheme applied
- purpose is to achieve scalable performance constrained by the aforementioned factors

Design Objectives of Computer Clusters

Packaging

- cluster nodes can be packaged in a compact or a slack fashion
- **compact cluster**
  - nodes are closely packaged in one or more racks sitting in a room
  - nodes are not attached to peripherals (monitors, keyboards, mice, etc.)
- **slack cluster**
  - nodes are attached to their usual peripherals (i.e., they are complete workstations, and PCs)
  - may be located in different rooms, different buildings, or even remote regions

Design Objectives of Computer Clusters

Control

- cluster can be managed in a centralized or decentralized fashion
- compact cluster normally has centralized control
- slack cluster can be controlled either way
- centralized cluster
  - nodes are owned, managed, and administered by a central operator
- decentralized cluster
  - nodes have individual owners
  - lacks a single point of control

Design Objectives of Computer Clusters

Homogeneity

- **homogeneous cluster**
  - uses nodes from the same platform, i.e., the same processor architecture and the same operating system
- **heterogeneous cluster**
  - uses nodes of different platforms
  - interoperability is an important issue

Fundamental Cluster Design Issues

Cluster Job Management

- achieve high system utilization
- job management software is required to provide batching, load balancing, parallel processing, and other functionality

Single-System Image (SSI)

- cluster is a single system
- appealing goal, very difficult to achieve
- SSI techniques are aimed at achieving this goal

Fundamental Cluster Design Issues

Availability Support

- redundancy in processors, memory, disks, I/O devices, networks, and operating system images

Fault Tolerance and Recovery

- eliminate all single points of failure
- tolerate faulty conditions up to a certain extent through redundancy
- critical jobs running on the failing nodes can be saved by failing over to the surviving node machines
- rollback recovery schemes for periodic checkpointing

Single-System Image

- **motivation** - allows a cluster to be used, controlled, and maintained as a familiar workstation

- **features:**
  - **Single system**
  - **Single control**
  - **Symmetry**
  - **Location-transparent**
  - **Single job management system**
  - **Single user interface**
  - **Single process space**

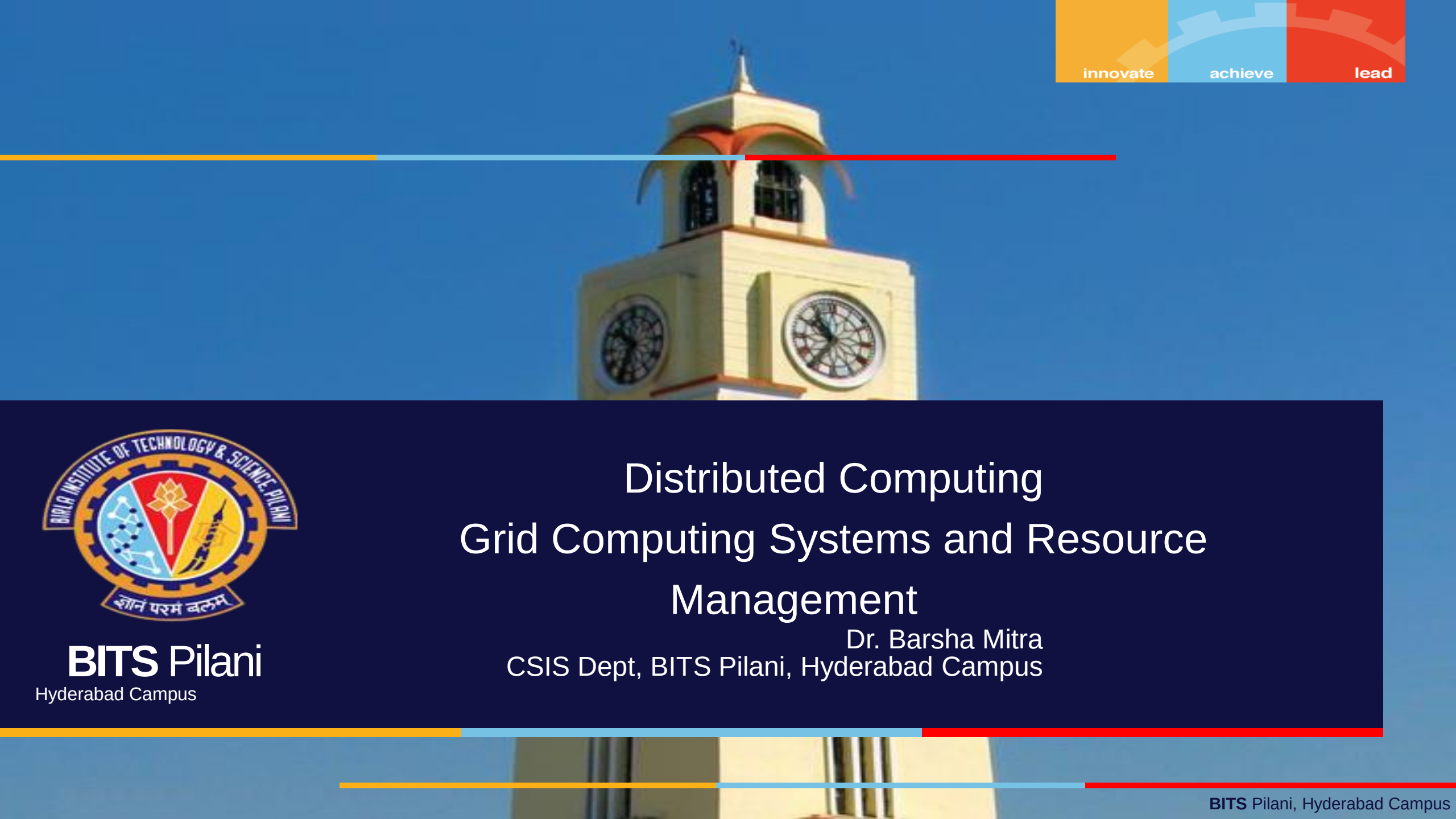
High Availability

Fault-Tolerant Cluster Configurations:

- **Hot standby server clusters**
  - only the primary node is actively doing all the useful work normally
  - standby node is powered on (hot) and run some monitoring programs to check the status of the primary node
- **Active-takeover clusters**
  - architecture is symmetric among multiple nodes
  - all nodes are primary, doing useful work normally
  - when a node fails, user applications switch to the available node in cluster

Reference

- Kai Hwang, Geoffrey C. Fox, and Jack J. Dongarra, "Distributed and Cloud Computing: From Parallel processing to the Internet of Things", Chapter 2.



innovate achieve lead

BITS Pilani
Hyderabad Campus

Distributed Computing
Grid Computing Systems and Resource Management

Dr. Barsha Mitra
CSIS Dept, BITS Pilani, Hyderabad Campus

Grid Architecture

- brings together computers (PCs, workstations, server clusters, supercomputers, laptops, notebooks)
- to form a large collection of compute, storage, and network resources
- for solving large-scale computation problems or to enable fast information retrieval
- leasing the hardware, software, middleware, databases, and networks

Grid Architecture

- **goal of grid computing** - explore fast solutions for large-scale computing problems
- takes advantage of existing computing resources scattered in a nation or internationally around the globe
- resources owned by different organizations are aggregated together and shared by many users in collective applications
- heavy use of LAN/WAN resources across enterprises, organizations, and governments

Grid Service Families

- **computational/data grids** –
  - most of today's grid systems are of this type
- **information/ knowledge grids** –
  - dedicated to knowledge management and distributed ontology processing
- **business grids** –
  - seen in business world
  - built for business data/information processing

CPU Scavenging

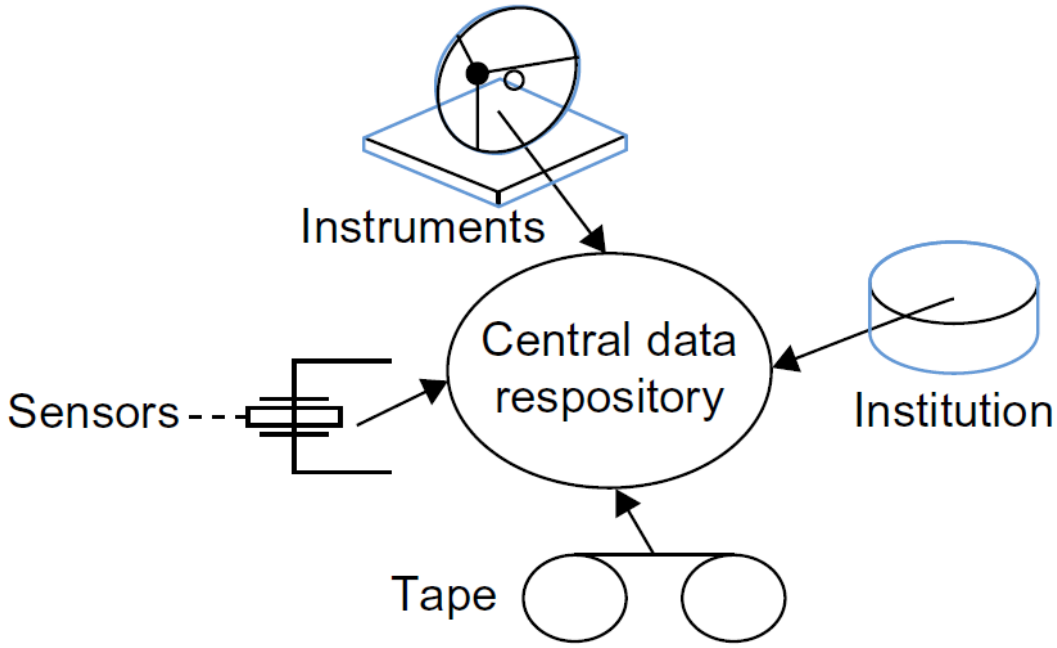
- concept of creating a “grid” from the unused resources in a network of computers
- grids are built over large number of desktop computers by using their free cycles at night or during inactive usage periods
- donors can be organizations or ordinary citizens on a voluntary participation basis
- client hosts also donate some disk space, RAM, and network bandwidth in addition to raw CPU cycles

Types of Applications in Grid

- computation-intensive
- data-intensive:
  - deal with massive amounts of data
  - grid system must be specially designed to discover, transfer, and manipulate massive data sets
  - efficient data management demands low-cost storage and high-speed data movement

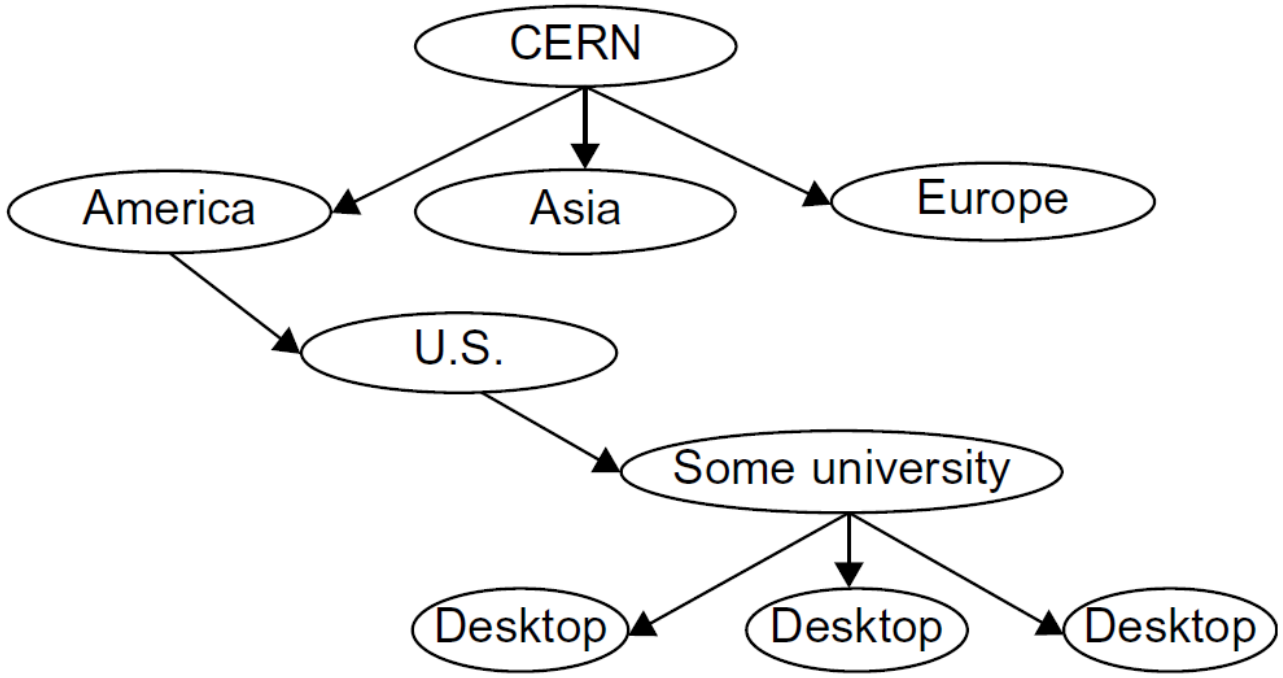
Grid Data Access Models

- **Monadic model:**
  - all data is saved in a central data repository
  - no data is replicated
  - simplest to implement for a small grid
  - not efficient for a large grid in terms of performance and reliability



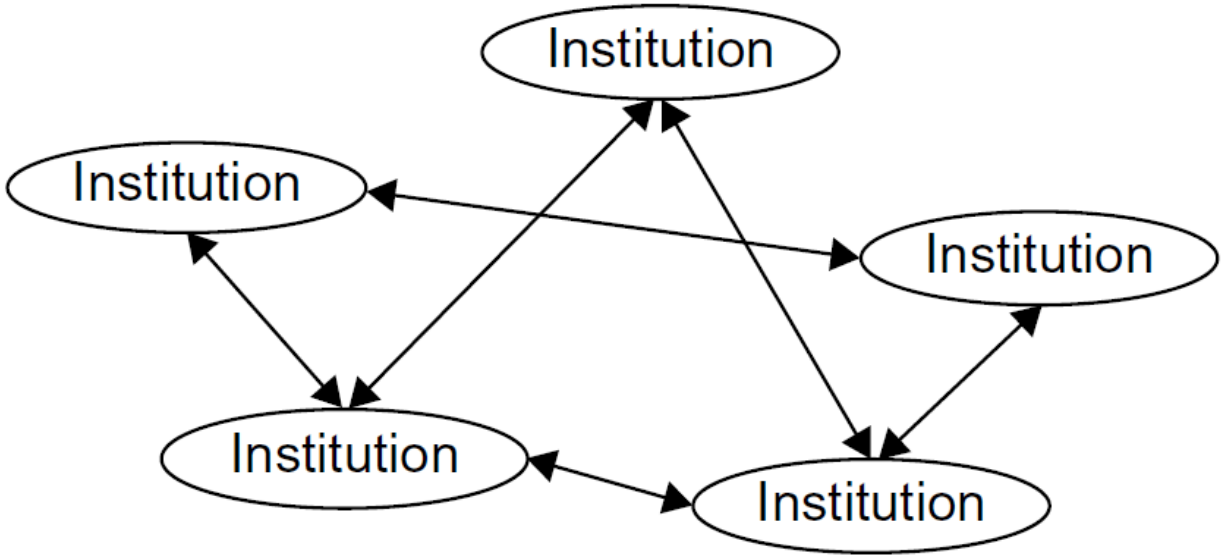
Grid Data Access Models

- **Hierarchical model:**
  - suitable for building a large data grid which has only one large data access directory
  - data is transferred through different centers at different levels



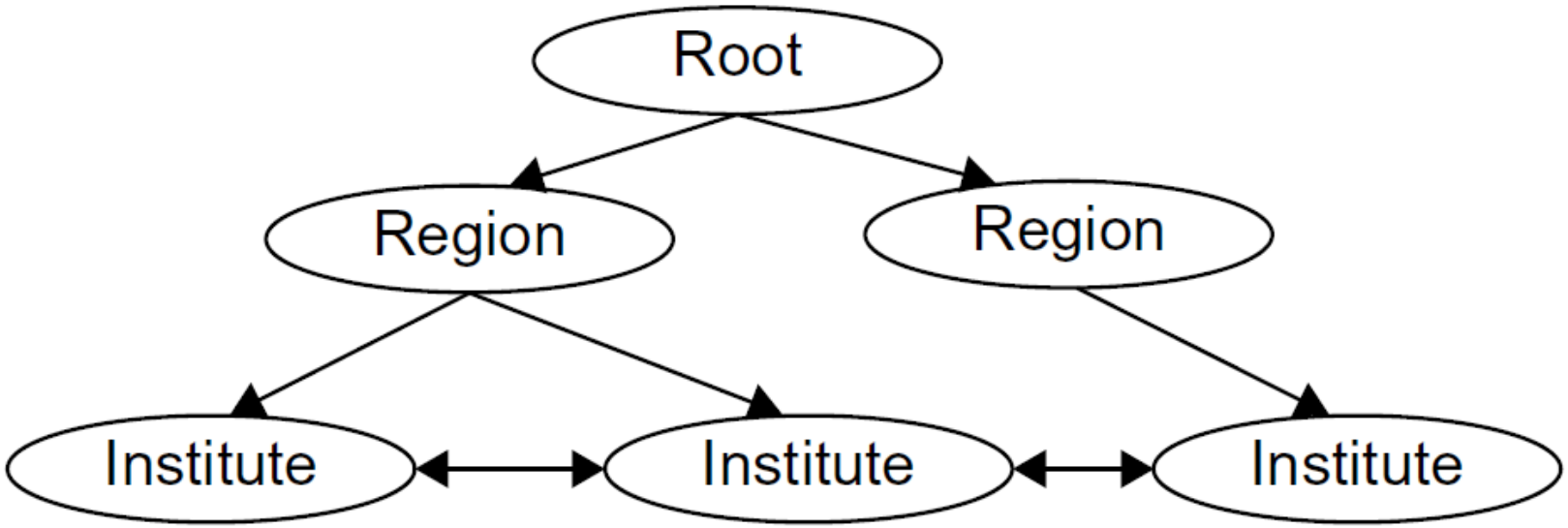
Grid Data Access Models

- **Federation model:**
  - data grid with multiple sources of data supplies
  - also known as a mesh model
  - data sources are distributed to different locations
  - data items are owned and controlled by their original owners
  - only authenticated users are authorized to request data from any data source
  - costly for large grids



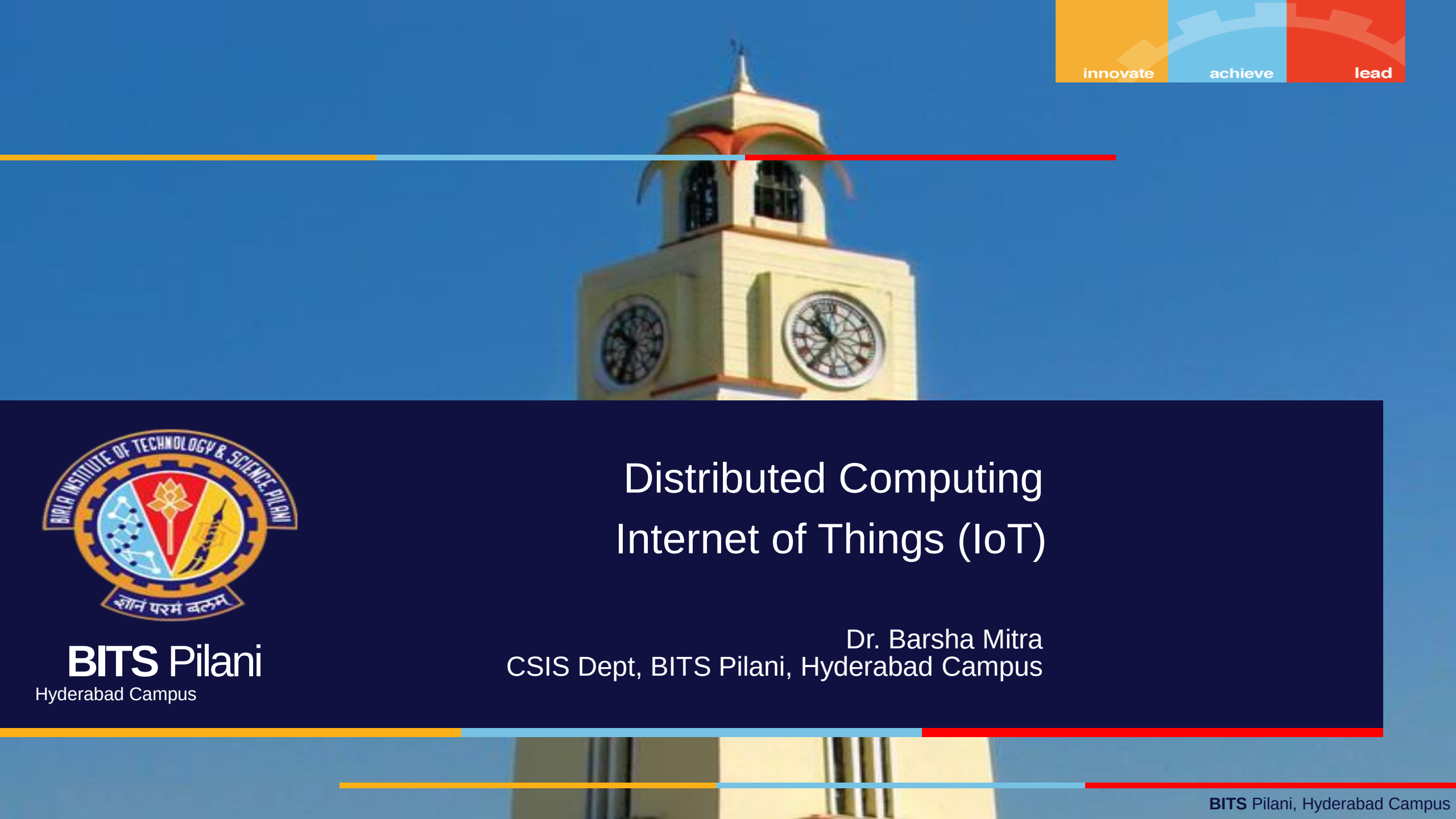
Grid Data Access Models

- **Hybrid model:**
  - combines the best features of hierarchical and mesh models
  - cost of the hybrid model can be traded off between the two extreme models for hierarchical and mesh-connected grids



Reference

- Kai Hwang, Geoffrey C. Fox, and Jack J. Dongarra, "Distributed and Cloud Computing: From Parallel processing to the Internet of Things", Chapter 7.



innovate achieve lead

BITS Pilani
Hyderabad Campus

Distributed Computing
Internet of Things (IoT)

Dr. Barsha Mitra
CSIS Dept, BITS Pilani, Hyderabad Campus

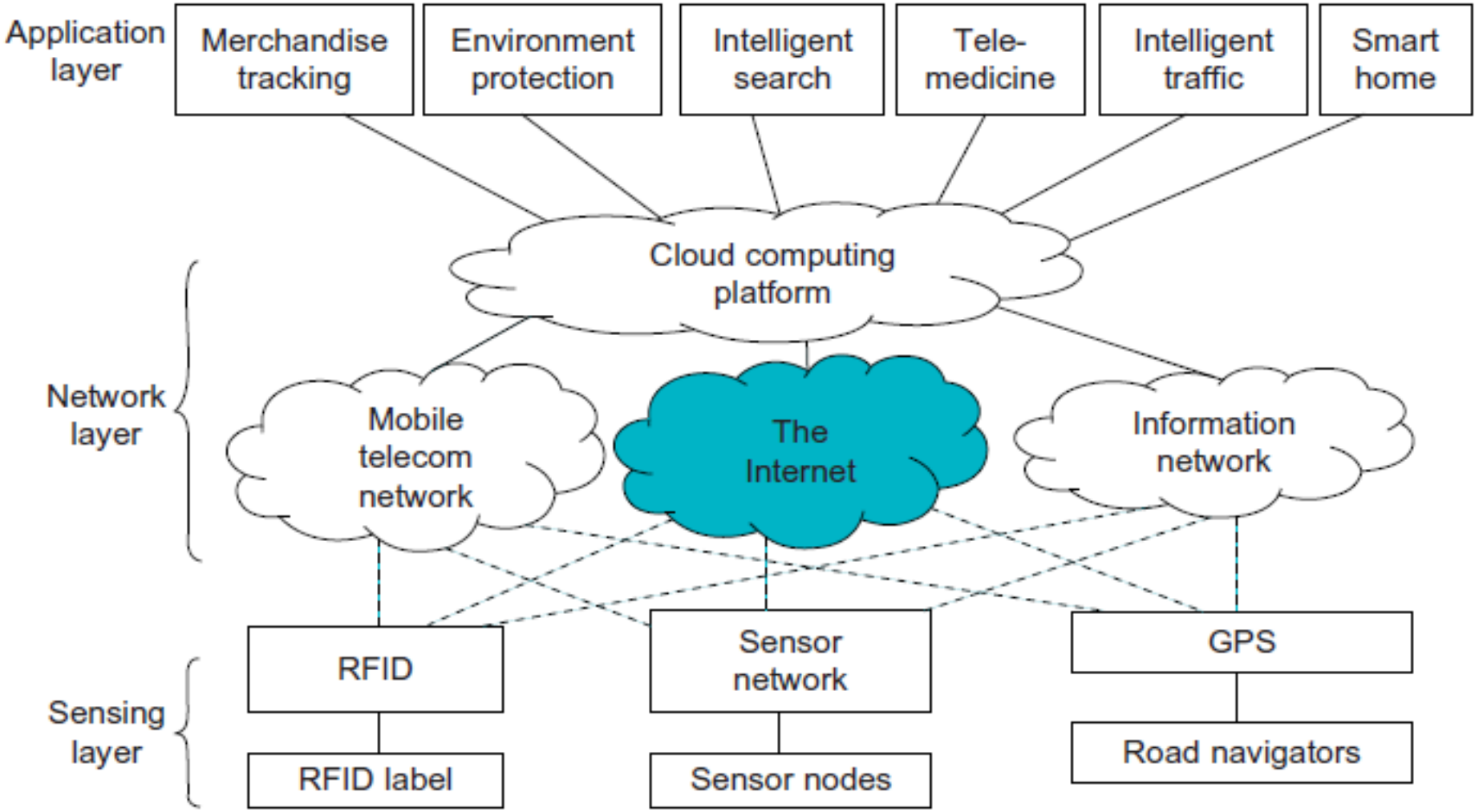
IoT for Ubiquitous Computing

- IoT is a natural extension of the Internet
- foundation of IoT is radio-frequency identification (RFID)
- RFID enables discovery of tagged objects and mobile devices by browsing an IP address
- objects include electronic devices, humans, animals, food, clothing, homes, vehicles, commodities, trees, hills, landmarks

Architecture of the Internet of Things

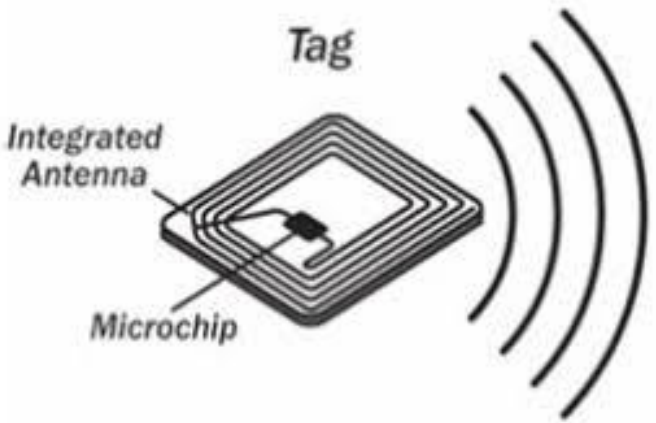
- 3-layer architecture
- **top layer:** consists of applications
- **bottom layers:** represent various types of sensing devices like RFID tags, ZigBee or other types of sensors, and road-mapping GPS navigators
- signals or information collected at these sensing devices are linked to the applications through the cloud computing platforms at the **middle layer**
- sensors and filters are used to collect raw data
- compute and storage clouds and grids process the data and transform it into information and knowledge formats
- sensed information is put together for a decision-making system

Architecture of the Internet of Things



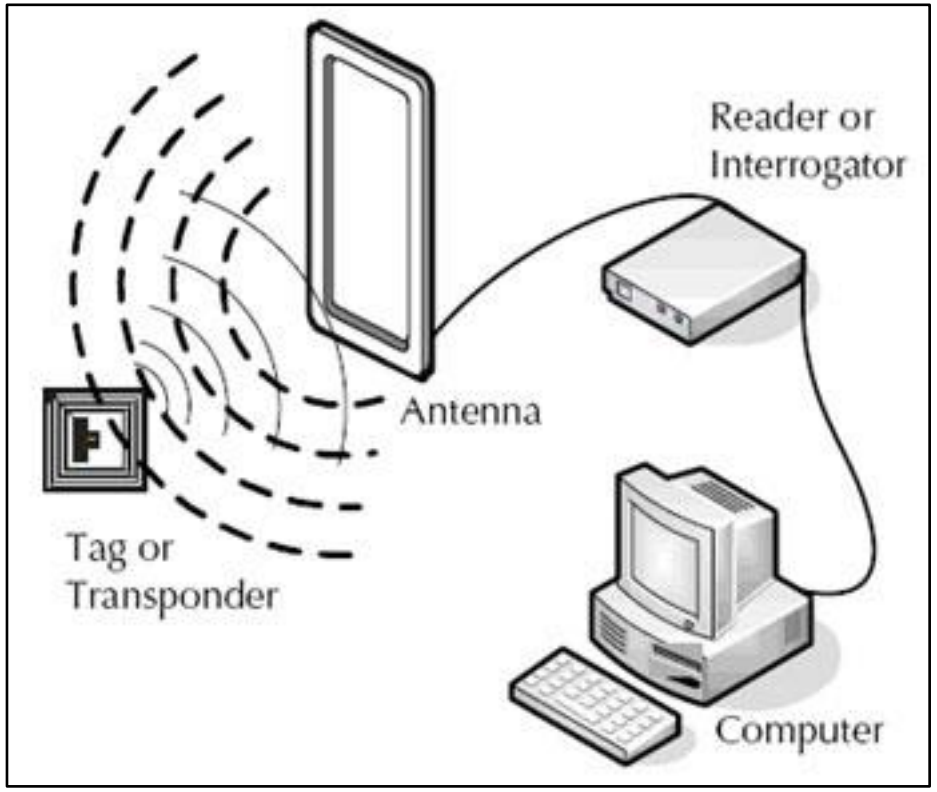
Radio-Frequency Identification (RFID)

- RFID is applied with electronic labels or RFID tags on any object being monitored or tracked
- identify and track objects using radio waves or sensing signals
- **Components of RFID h/w**
  - **RFID tag** - tiny silicon chip attached to a small antenna
  - 2 major parts
    - **integrated circuit** for storing and processing information, modulating and demodulating radio-frequency (RF) signal etc.
    - **antenna** for receiving and transmitting radio signals



Radio-Frequency Identification (RFID)

- **Components of RFID h/w**
  - **Reader antenna** –
    - used to radiate energy and then capture the return signal sent back from the tag
    - integrated with a handheld reader device or connected to the reader by cable
  - **Reader** –
    - device station that talks with the tags
    - may support one or more antennae



Types of RFID Tags

- **active RFID tags** containing a battery and transmitting signals autonomously
- **passive RFID tags** have no battery and require an external source to induce signal transmission
- **battery-assisted passive RFID tags** require an external source to wake up the battery

References

- Kai Hwang, Geoffrey C. Fox, and Jack J. Dongarra, “Distributed and Cloud Computing: From Parallel processing to the Internet of Things”, Chapter 9.
- https://www.exchangecommunications.co.uk/products/smart-building-and-cities/smart-buildings.php
- https://blog.econocom.com/en/blog/smartbuilding-and-bms-a-little-glossary/
- https://www.security-warehouse.com/rfid-iot/rfid-antenna.html
- https://www.scnsoft.com/blog/iot-for-inventory-management
- https://www.autodesk.com/products/eagle/blog/rfid-works-antenna-design/

Thank You