RAMAIAH
Institute of Technology

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

Course Name: Distributed Systems

Course Code – CS751

Credits - 3:0:0

UNIT -1

Term: JULY 2023 – NOV 2023

Prepared by: Dr. Sangeetha. V
Associate Professor

RAMAIAH Institute of Technology

Textbooks

Textbook:

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

Reference Books:

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

RAMAIAH Institute of Technology

Other CIE Component(20 marks)

NPTEL: Distributed Systems, IIT Patna

Dr. Rajiv Misra

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

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

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

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

Date of exam: September 24, 2023

RAMAIAH Institute of Technology

UNIT -1

Introduction:

(Chapter 1-1.1,1.2,1.3,1.4,1.5,1.6,1.7,1.8)

- Definition
- Relation to computer system components
- Motivation
- Relation to parallel multiprocessor/multicomputer systems,
- Message-passing systems versus shared memory systems.
- Primitives for distributed communication
- Synchronous versus asynchronous executions
- Design issues and challenges

A model of distributed computations:

(Chapter 2 – 2.1,2.2,2.3,2.4,2.5,2.6)

- A distributed program
- A model of distributed executions
- Models of communication networks
- Global state of a distributed system
- Cuts of a distributed computation
- Past and future cones of an event

Logical time: (Chapter 3: 3.1,3.2,3.3,3.4,3.5,3.6)

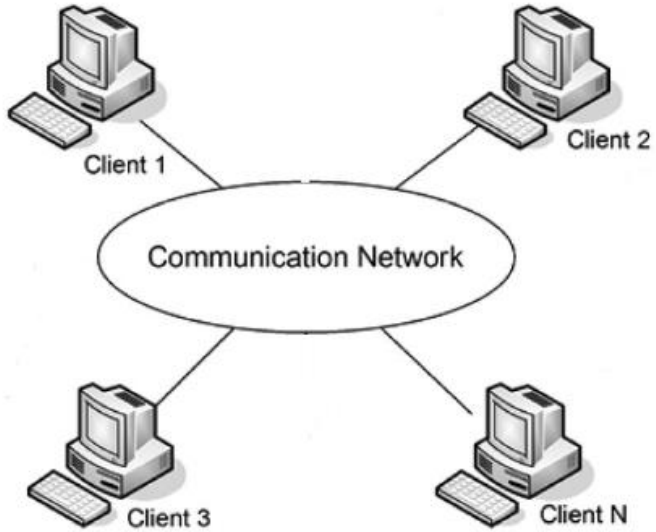
- Introduction
- A framework for a system of logical clocks
- Scalar time
- Vector time
- efficient implementations of vector clocks
- Jard–Jourdan's adaptive technique

RAMAIAH Institute of Technology

Introduction

A centralized computing system is where all computing is performed by a single computer in one location.

A distributed system is a collection of independent computers, interconnected via a network, capable of collaborating on a task.



Example

Banking systems are distributed systems.

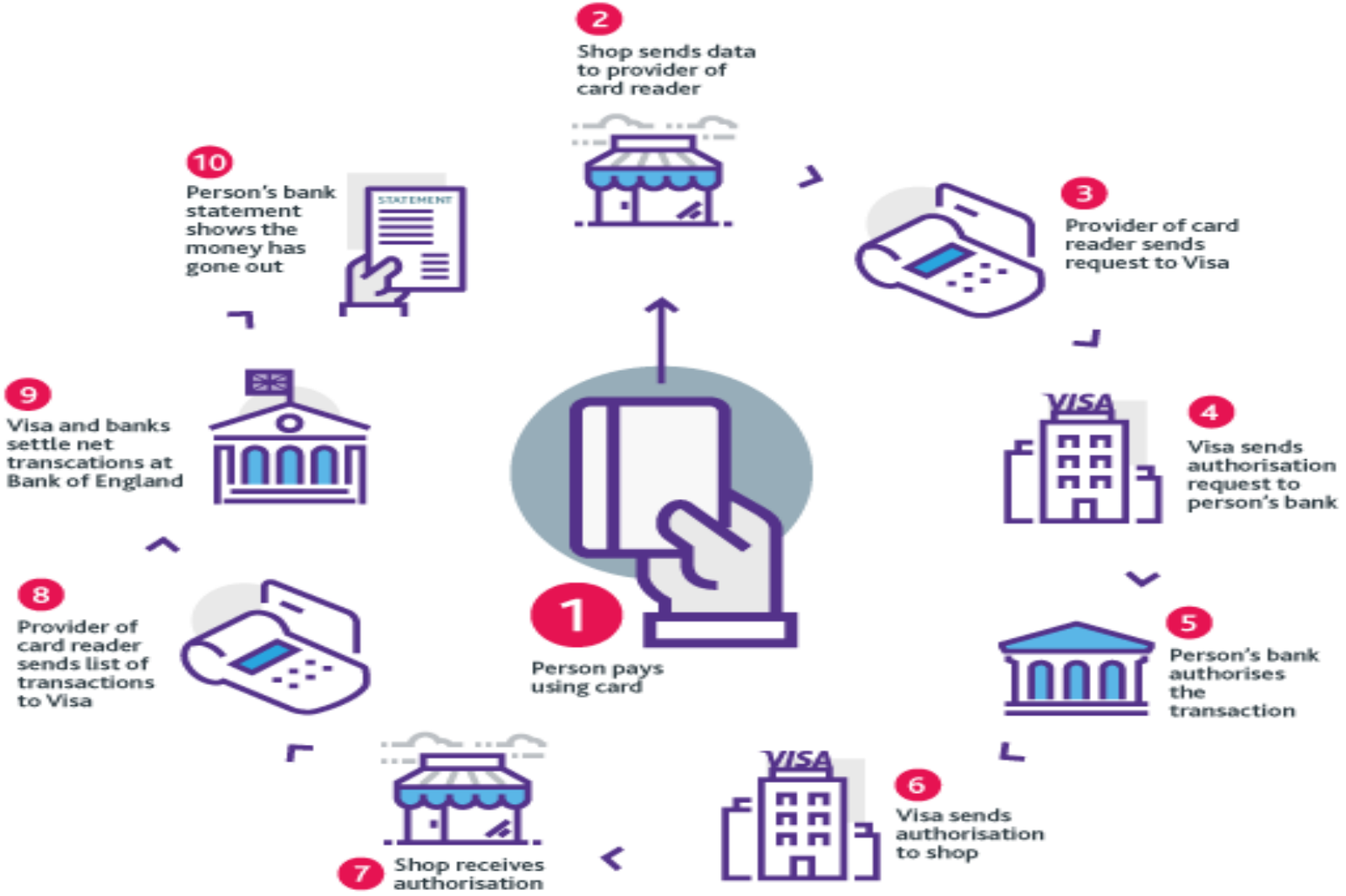
There are systems that are

- handling the actual transactions against your checking/saving accounts,
- that deal with the processing of a credit or debit-card transaction,
- systems that deal with mortgages/loans,

yet they are all operating on your accounts.

RAMAIAH
Institute of Technology

Using a debit card to make a purchase



Applications of DS

Finance and Commerce: Amazon, eBay, Online Banking, E-Commerce websites.

Information Society: Search Engines, Wikipedia, Social Networking, Cloud Computing.

Entertainment: Online Gaming, Music, youtube.

Healthcare: Online patient records, Health Informatics.

Education: E-learning.

Transport and logistics: GPS, Google Maps.

RAMAIAH Institute of Technology

UNIT -1

Introduction:(Chapter 1-1.1,1.2,1.3,1.4,1.5,1.6,1.7,1.8)

- Definition
- Relation to computer system components
- Motivation
- Relation to parallel multiprocessor/multicomputer systems,
- Message-passing systems versus shared memory systems.
- Primitives for distributed communication
- Synchronous versus asynchronous executions
- Design issues and challenges

Definition

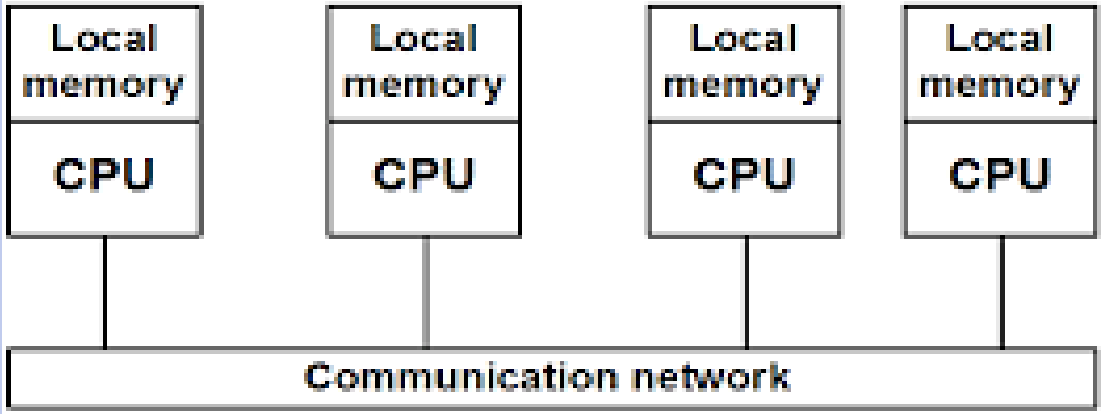
A distributed system is a collection of independent entities that cooperate to solve a problem that cannot be individually solved.

A distributed system has been characterized in one of several ways:

- You know you are using one when the crash of a computer you have never heard of prevents you from doing work .
- A collection of computers that do not share common memory or a common physical clock, that communicate by a messages passing over a communication network, and where each computer has its own memory and runs its own operating system. Typically the computers are semi-autonomous and are loosely coupled while they cooperate to address a problem collectively

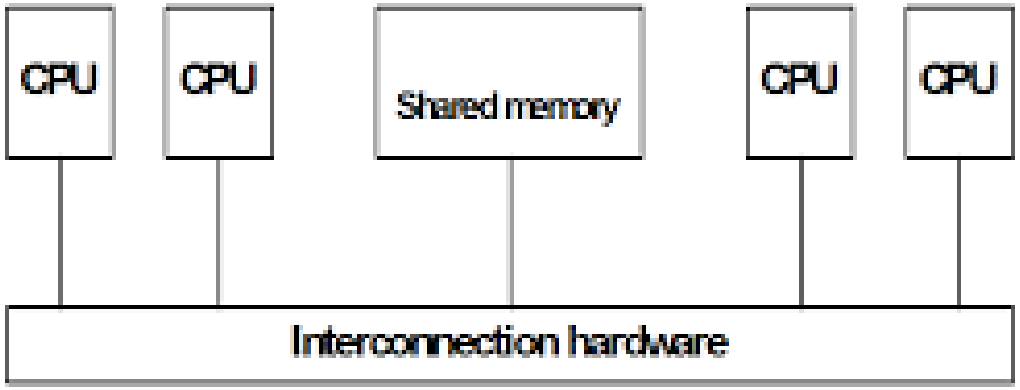
Loosely Coupled System

- Distributed Memory Systems (DMS)
- Communication via Message Passing



Tightly Coupled Systems

- Systems with a single system wide memory
- Parallel Processing System , SMMP (shared memory multiprocessor systems)



Definition

- A collection of independent computers that appears to the users of the system as a single coherent computer
- A term that describes a wide range of computers,
  - weakly coupled systems such as wide-area networks
  - strongly coupled systems such as local area networks,
  - strongly coupled systems such as multiprocessor systems

Definition – DS Features

No common physical clock - This is an important assumption because it introduces the element of "distribution" in the system and gives rise to the inherent asynchrony amongst the processors.

No shared memory- provide the abstraction of a common address space via the distributed shared memory abstraction.

Definition – DS Features

Geographical separation

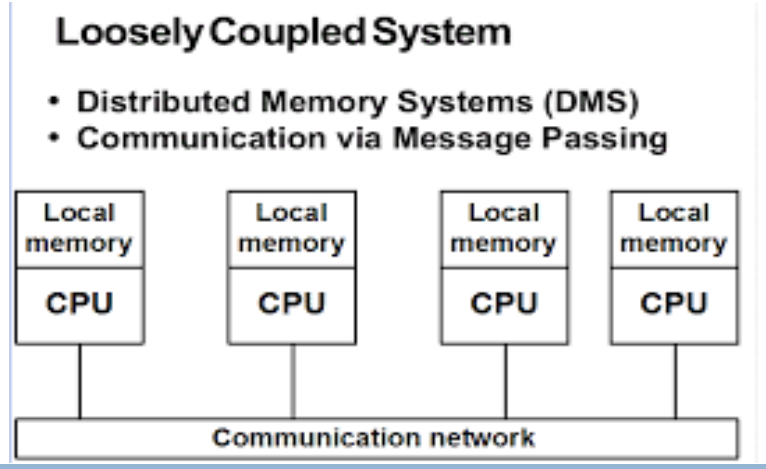
- Processors to be on a wide-area network (WAN).
- The network/cluster of workstations (NOW/COW) configuration connecting processors on a LAN is also being increasingly regarded as a small distributed system.
- This NOW configuration is becoming popular because of the low-cost high-speed off-the-shelf processors now available.
- The Google search engine is based on the NOW architecture.

Definition – DS Features

Autonomy and heterogeneity

The processors are “loosely coupled” in that they have different speeds and each can be running a different operating system.

They are usually not part of a dedicated system, but cooperate with one another by offering services or solving a problem jointly.



Relation to computer system components

A typical distributed system is shown in Figure 1.1.

Each computer has a memory-processing unit and the computers are connected by a communication network.

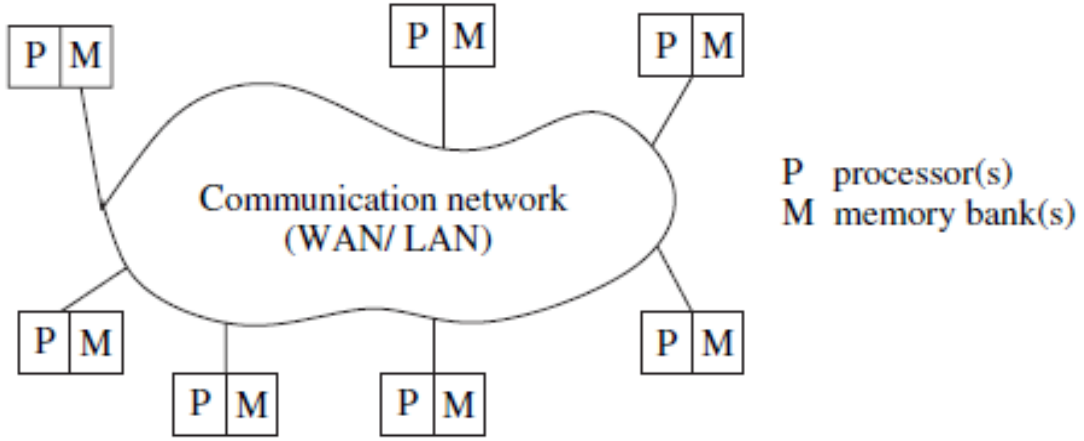


Figure 1.1 A distributed system connects processors by a communication network.

Relation to computer system components

- Figure 1.2 shows the relationships of the software components that run on each of the computers and use the local operating system and network protocol stack for functioning.
- The distributed software is also termed as middleware.
- A distributed execution is the execution of processes across the distributed system to collaboratively achieve a common goal.
- An execution is also sometimes termed a computation or a run.

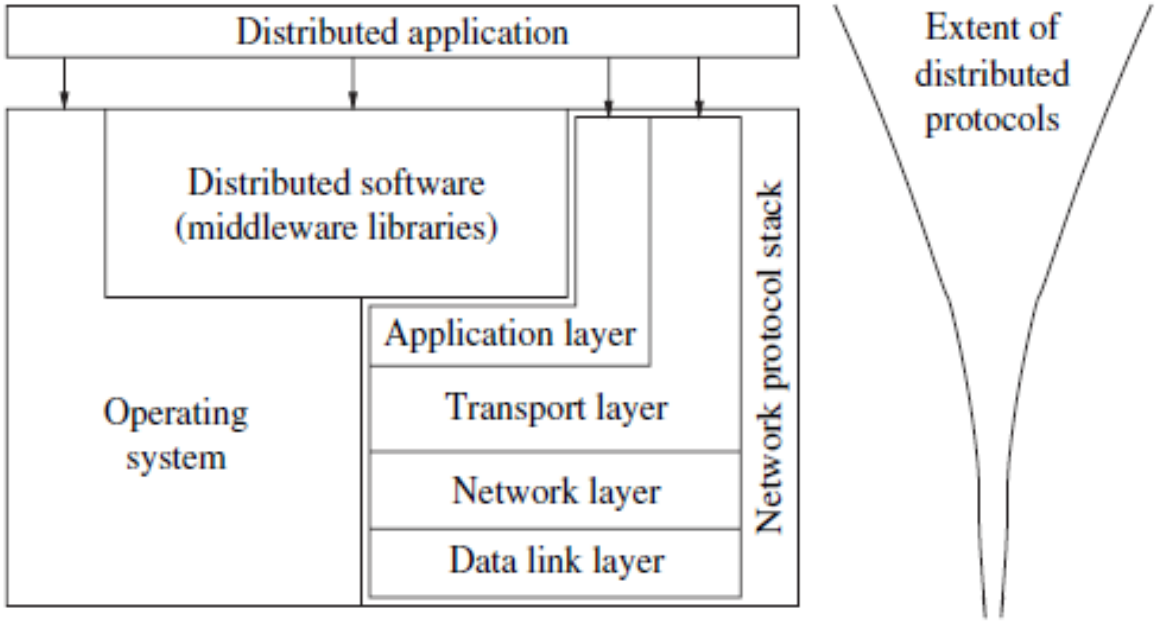


Figure 1.2 Interaction of the software components at each processor.

Relation to computer system components

The middleware is the distributed software that drives the distributed system, while providing transparency of heterogeneity at the platform level

Here we assume that the middleware layer does not contain the traditional application layer functions of the network protocol stack, such as *http, mail, ftp, and telnet.*

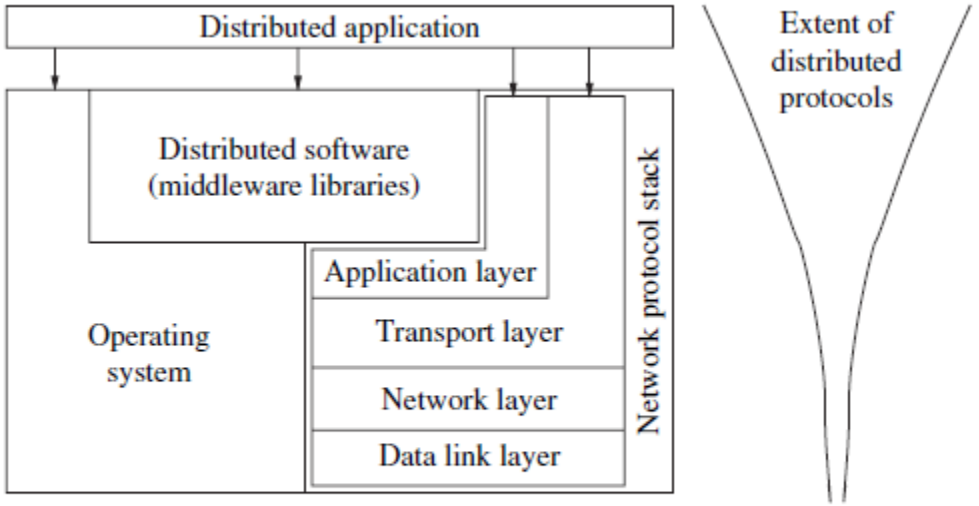


Figure 1.2 Interaction of the software components at each processor.

Relation to computer system components

Various primitives and calls to functions defined in various libraries of the middleware layer are embedded in the user program code.

Several standards

- Object Management Group's (OMG)
- Common object request broker architecture (CORBA)
- Remote procedure call (RPC)

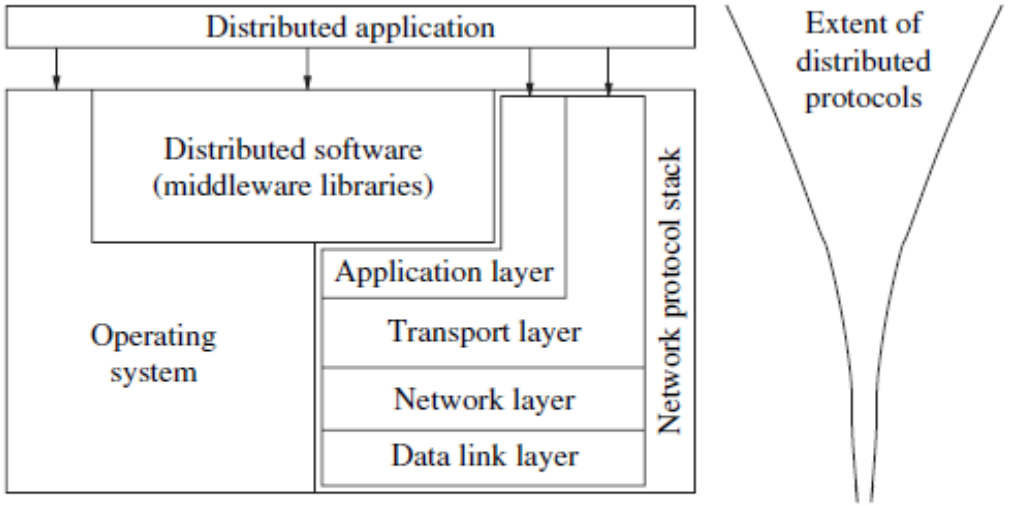


Figure 1.2 Interaction of the software components at each processor.

Motivation

- **Inherently distributed computations**

  In many applications such as money transfer in banking, or reaching consensus among parties that are geographically distant, the computation is inherently distributed.

Motivation

- **Resource sharing**

  Resources such as peripherals, complete data sets in databases, special libraries, as well as data (variable/files) cannot be fully replicated at all the sites because it is often neither practical nor cost-effective.

  Further, they cannot be placed at a single site because access to that site might prove to be a bottleneck.

  Therefore, such resources are typically distributed across the system.

  For example, distributed databases such as DB2 partition the data sets across several servers, in addition to replicating them at a few sites for rapid access as well as reliability.

Motivation

- **Access to geographically remote data and resources**

In many scenarios, the data cannot be replicated at every site participating in the distributed execution because it may be too large or too sensitive to be replicated.

For example, payroll data within a multinational corporation is both too large and too sensitive to be replicated at every branch office/site.

It is therefore stored at a central server which can be queried by branch offices.

Motivation

- **Enhanced reliability**

System has the inherent potential to provide increased reliability because of the possibility of replicating resources and executions, as well as the reality that geographically distributed resources are not likely to crash/malfunction at the same time under normal circumstances.

Reliability entails several aspects:

- availability, i.e., the resource should be accessible at all times;
- integrity, i.e., the value/state of the resource should be correct, in the face of concurrent access from multiple processors, as per the semantics expected by the application;
- fault-tolerance, i.e., the ability to recover from system failures,

RAMAIAH Institute of Technology

Motivation

- **Increased performance/cost ratio**

By resource sharing and accessing geographically remote data and resources, the performance/cost ratio is increased.

- **Scalability**

As the processors are usually connected by a wide-area network, adding more processors does not pose a direct bottleneck for the communication network.

- **Modularity and incremental expandability**

Heterogeneous processors may be easily added into the system without affecting the performance, as long as those processors are running the same middleware algorithms.

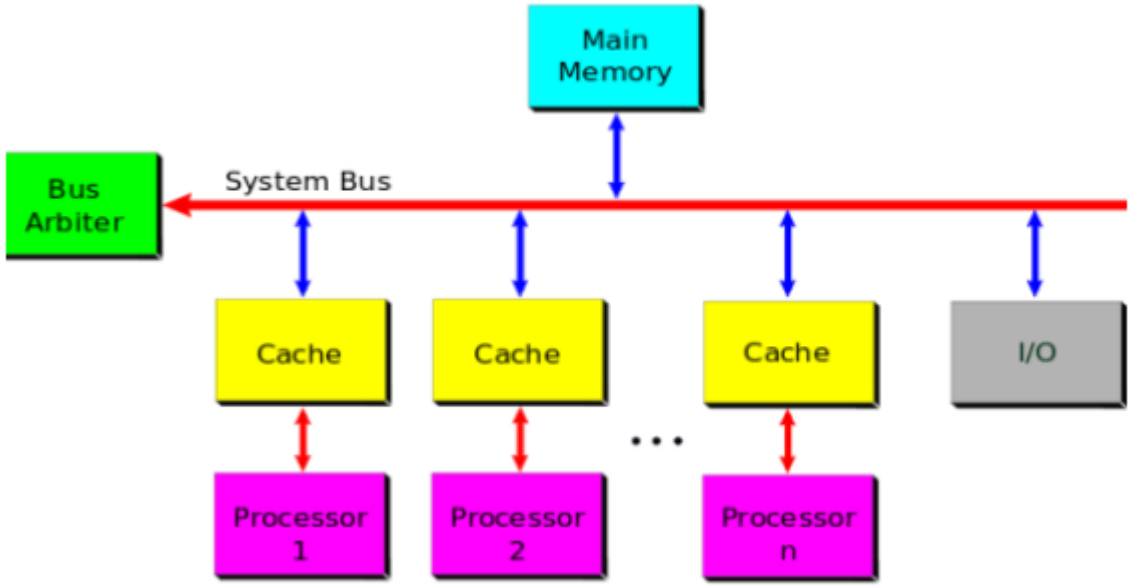
Relation to parallel multiprocessor/multicomputer systems

How to classify a system that meets some but not all of the characteristics?

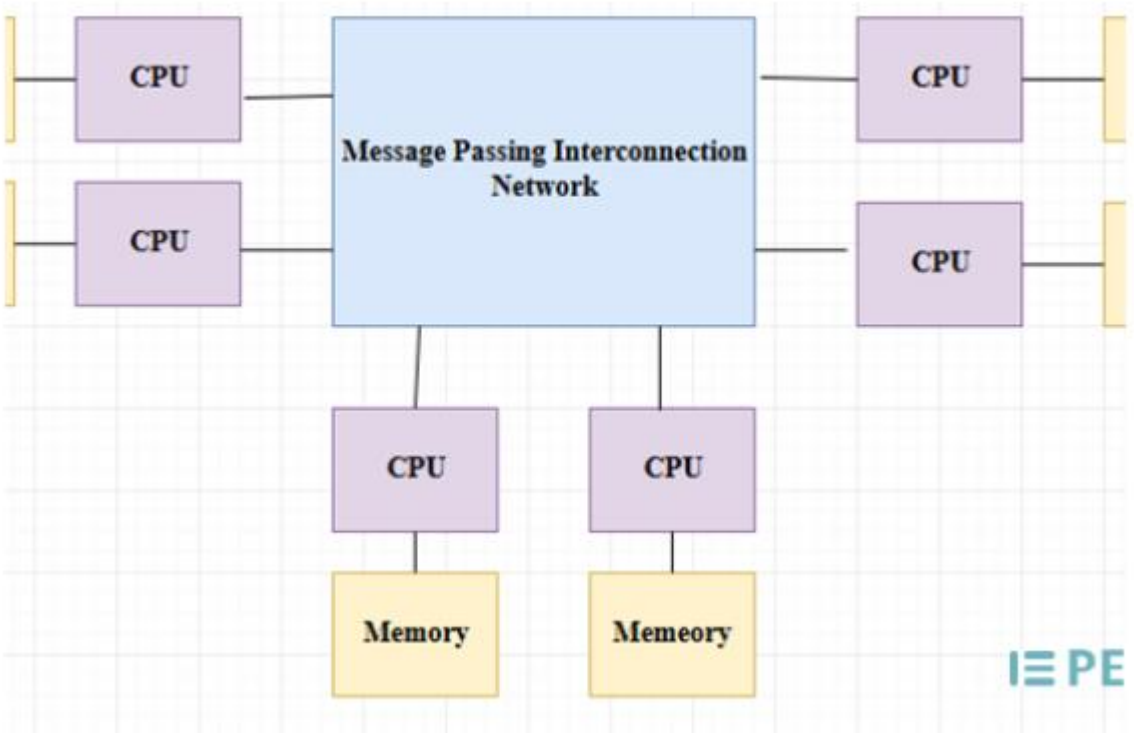
Is the system still a distributed system, or does it become a parallel multiprocessor system?

To better answer these questions, we first examine the architecture of parallel systems, and then examine some well-known taxonomies for multiprocessor/multicomputer systems.

Multiprocessor



Multicomputer



Relation to parallel multiprocessor/multicomputer systems

Characteristics of parallel systems

A parallel system is classified into 3 types:

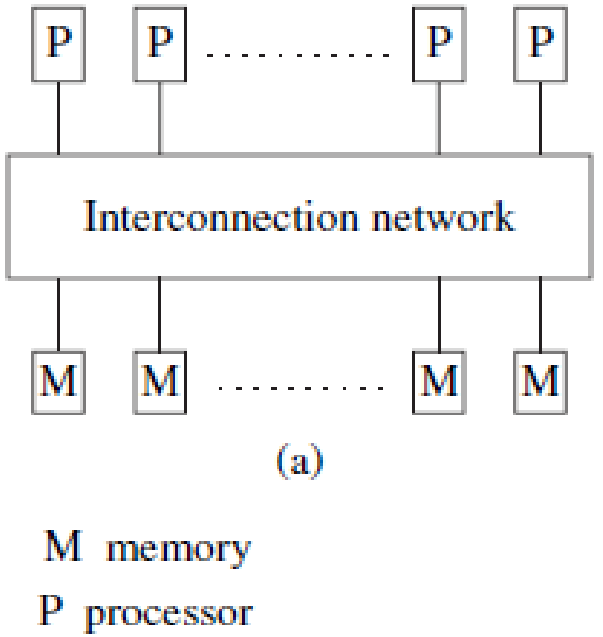
1. Multiprocessor system- Uniform memory access (UMA) architecture
2. Multicomputer parallel system- Non-uniform memory access (NUMA) multiprocessor
3. Array processors

Relation to parallel multiprocessor/multicomputer systems

1.Multiprocessor system

A multiprocessor system is a parallel system in which the multiple processors have direct access to shared memory which forms a common address space.

- Such processors usually do not have a common clock.
- A multiprocessor system usually corresponds to a uniform memory access (UMA) architecture in which the access latency, i.e., waiting time, to complete an access to any memory location from any processor is the same.

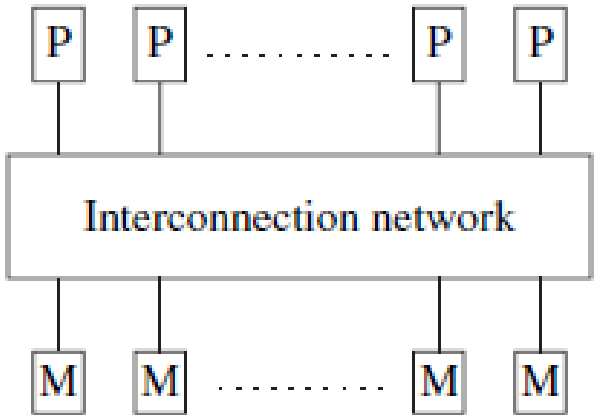


Relation to parallel multiprocessor/multicomputer systems

1.Multiprocessor system

The processors are in very close physical proximity and are connected by an interconnection network.

- Interprocess communication across processors is traditionally through read and write operations on the shared memory
- All the processors usually run the same operating system, and both the hardware and software are very tightly coupled.



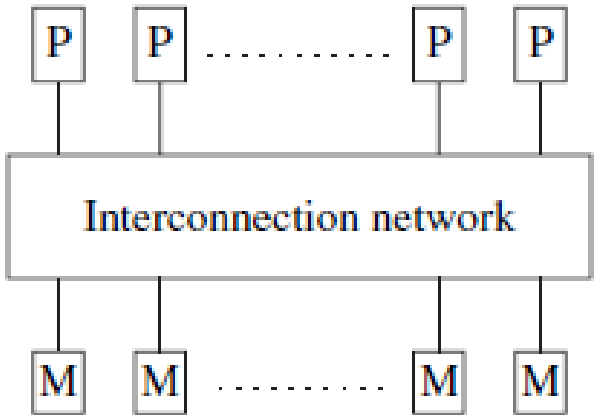
(a)

M memory
P processor

Relation to parallel multiprocessor/multicomputer systems

1.Multiprocessor system

The interconnection network to access the memory may be a bus, although for greater efficiency, it is usually a multistage switch with a symmetric and regular design.



(a)

M memory

P processor

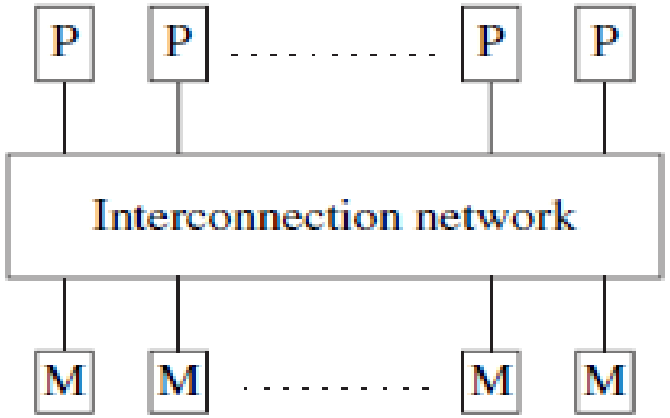
Relation to parallel multiprocessor/multicomputer systems

1.Multiprocessor system

Two popular inter connection networks

1. Omega network
2. Butterfly network

each of which is a multi-stage network formed of 2×2 switching elements.

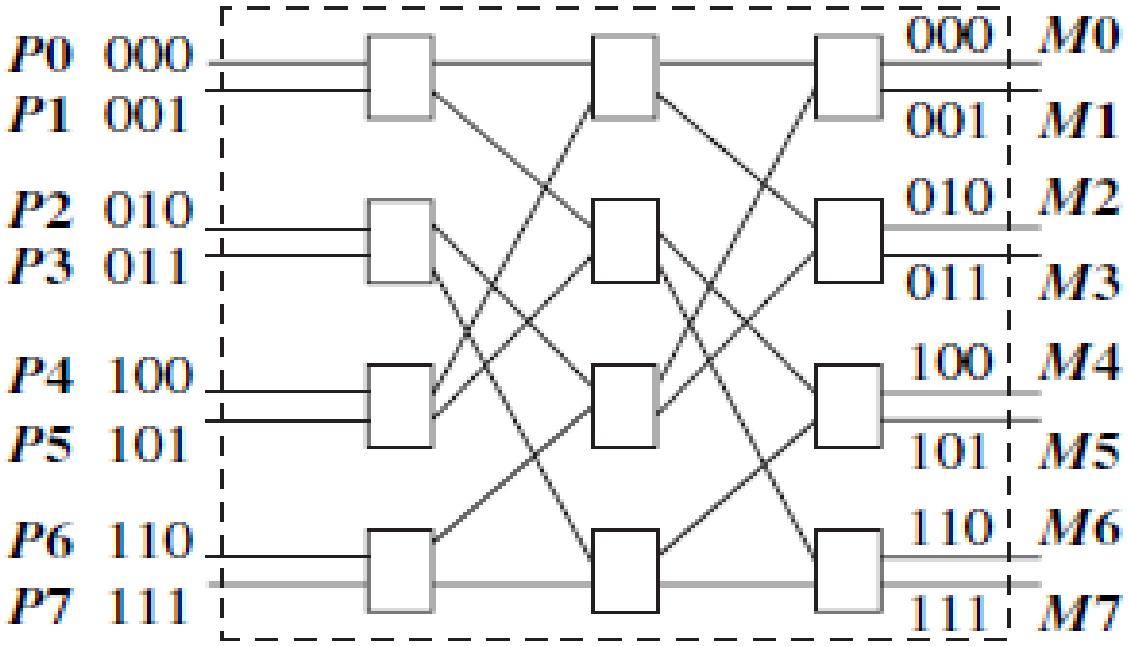


(a)

M memory
P processor

Relation to parallel multiprocessor/multicomputer systems

1.Multiprocessor system -Omega network

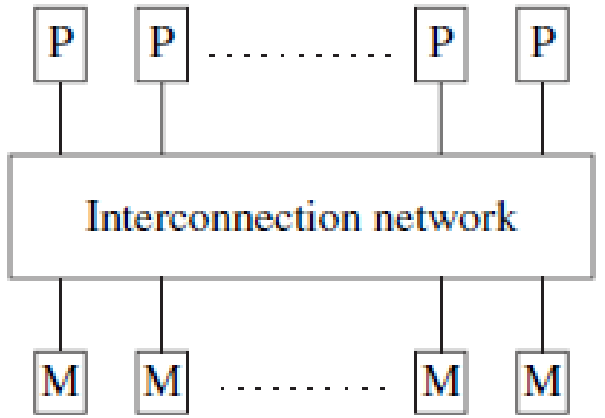


(a) 3-stage Omega network ($n = 8$, $M = 4$)

Omega network for
n = 8 processors P0–P7
memory banks M0–M7.

3-stage network

M = n/2
= 8/2
= 4 switches per stage



(a)

M memory
P processor

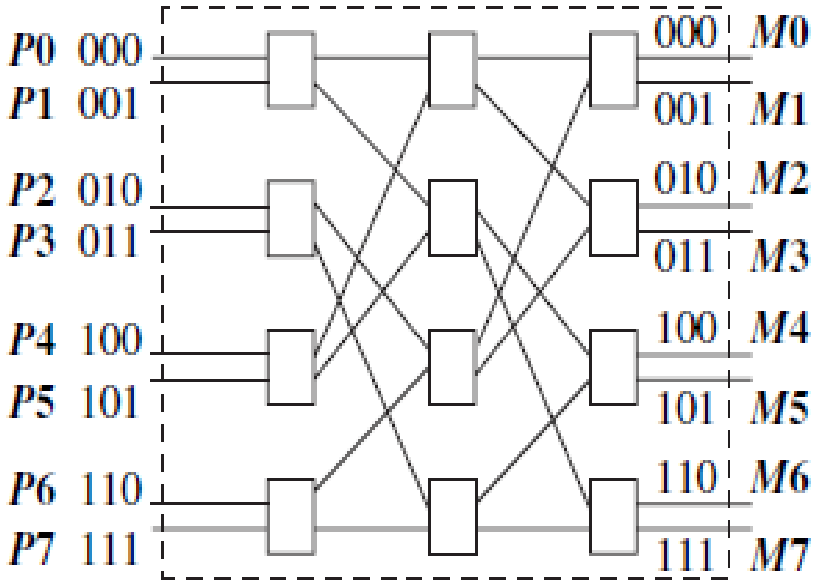
Relation to parallel multiprocessor/multicomputer systems

1.Multiprocessor system -Omega network

Each 2×2 switch is represented as a rectangle in the figure.

Each 2×2 switch allows data on either of the two input wires to be switched to the upper or the lower output wire.

In a single step, however, only one data unit can be sent on an output wire.

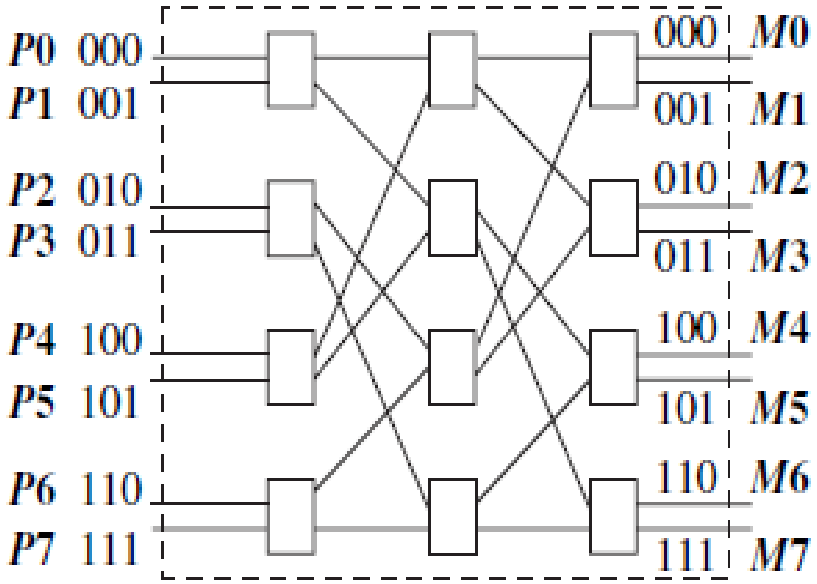


(a) 3-stage Omega network ($n = 8$, $M = 4$)

Relation to parallel multiprocessor/multicomputer systems

1.Multiprocessor system -Omega network

- A n-input and n-output network uses log n stages and log n bits for addressing.
- Routing in the 2×2 switch at stage k uses only the k^{th}bit, and hence can be done at clock speed in hardware.

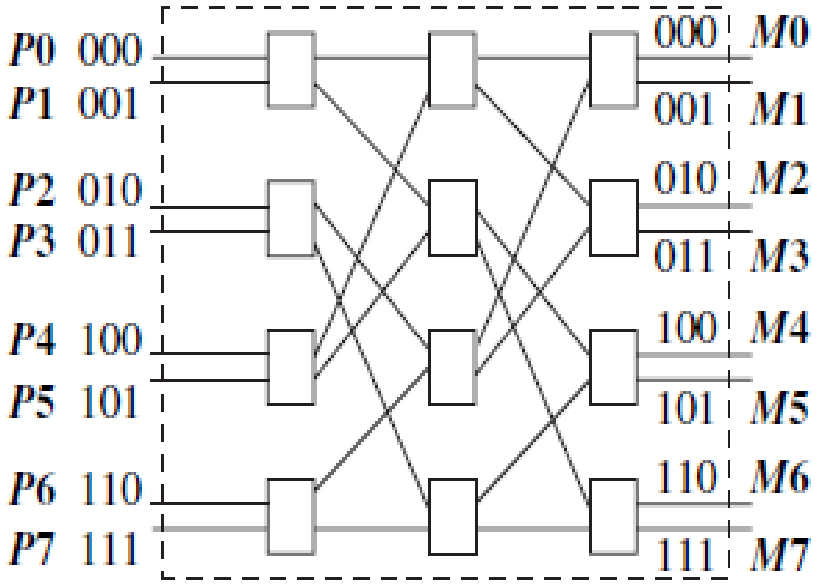


(a) 3-stage Omega network ($n = 8$, $M = 4$)

Relation to parallel multiprocessor/multicomputer systems

1.Multiprocessor system -Omega interconnection function

The Omega network which connects n processors to n memory units has n/2 $\log_2$ n switching elements of size 2×2 arranged in $\log_2$ n stages.

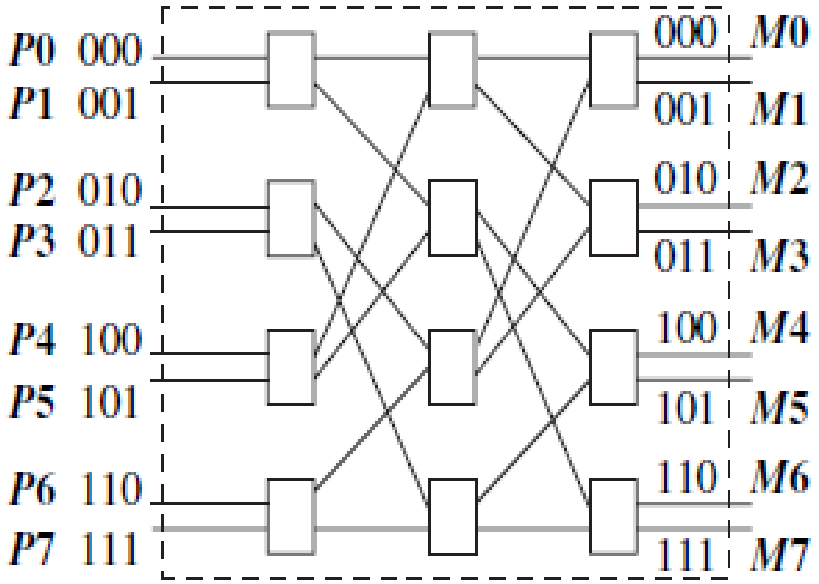


(a) 3-stage Omega network ($n = 8$, $M = 4$)

Relation to parallel multiprocessor/multicomputer systems

1.Multiprocessor system -Omega interconnection function

Between each pair of adjacent stages of the Omega network, a link exists between output i of a stage and the input j to the next stage according to the following **perfect shuffle pattern** which is a left-rotation operation on the binary representation of i to get j.



(a) 3-stage Omega network ($n = 8$, $M = 4$)

$$j = \begin{cases} 2i, & \text{for } 0 \le i \le n/2 - 1, \\ 2i + 1 - n, & \text{for } n/2 \le i \le n - 1. \end{cases}$$

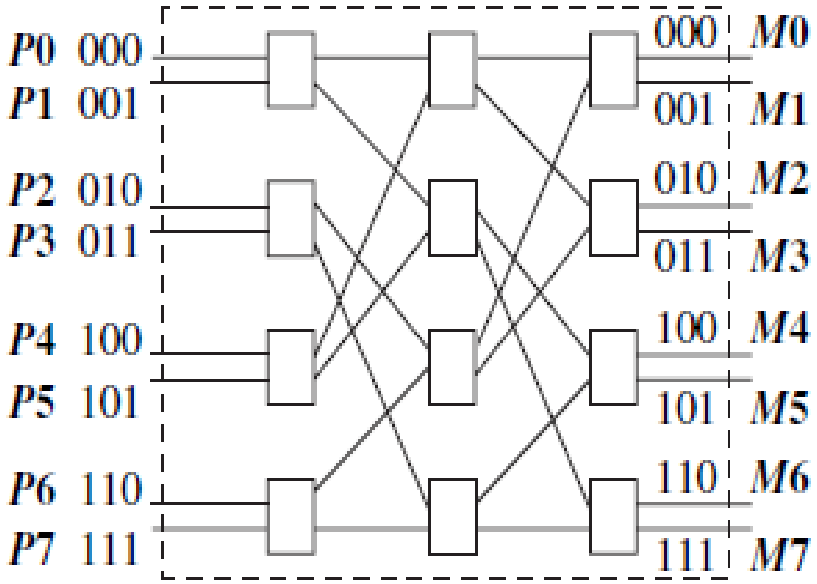
where $0 \le i \le 3$

where $4 \le i \le 7$

Relation to parallel multiprocessor/multicomputer systems

1.Multiprocessor system -Omega interconnection function

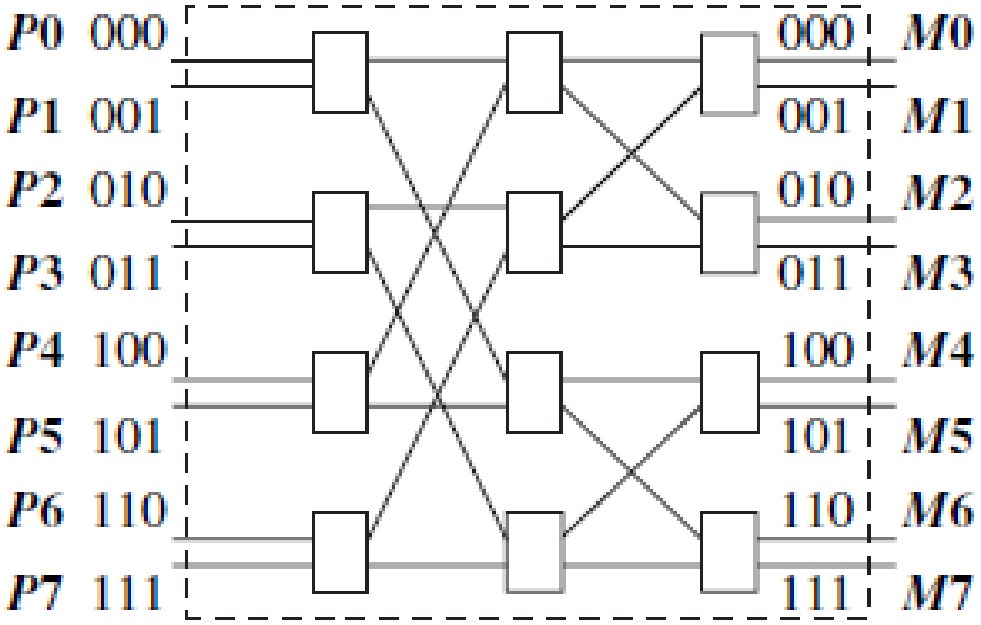
The Omega network which connects n processors to n memory units has n/2 $\log_2$ n switching elements of size 2×2 arranged in $\log_2$ n stages.



(a) 3-stage Omega network ($n = 8$, $M = 4$)

Relation to parallel multiprocessor/multicomputer systems

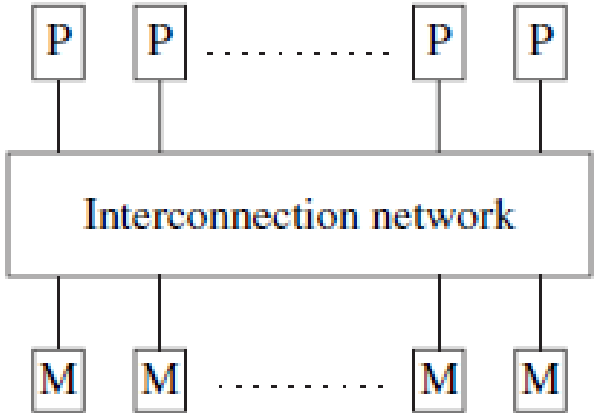
1.Multiprocessor system -Butterfly network



(b) 3-stage Butterfly network ($n = 8$, $M = 4$)

Butterfly network for
n = 8 processors P0–P7
memory banks M0–M7.

M = n/2
= 8/2
= 4 switches per stage



(a)

M memory
P processor

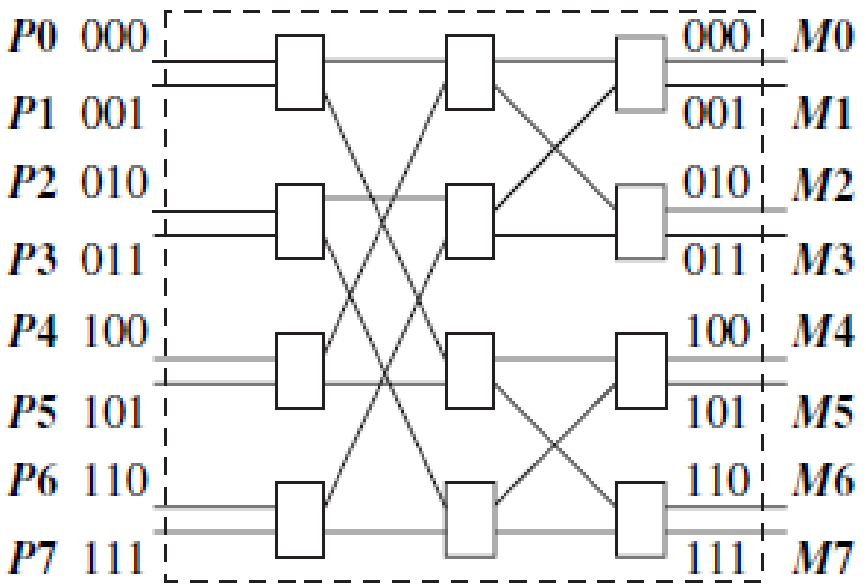
Relation to parallel multiprocessor/multicomputer systems

1.Multiprocessor system -Butterfly network

The generation of the interconnection pattern between a pair of adjacent stages depends not only on n but also on the stage number s.

Let there be M =n/2 switches per stage,

Let a switch be denoted by the tuple <x s>, where x ∈ [0, M−1] and stage s ∈ [0 ,$\log_2 n-1$].



(b) 3-stage Butterfly network ($n = 8$, $M = 4$)

Relation to parallel multiprocessor/multicomputer systems

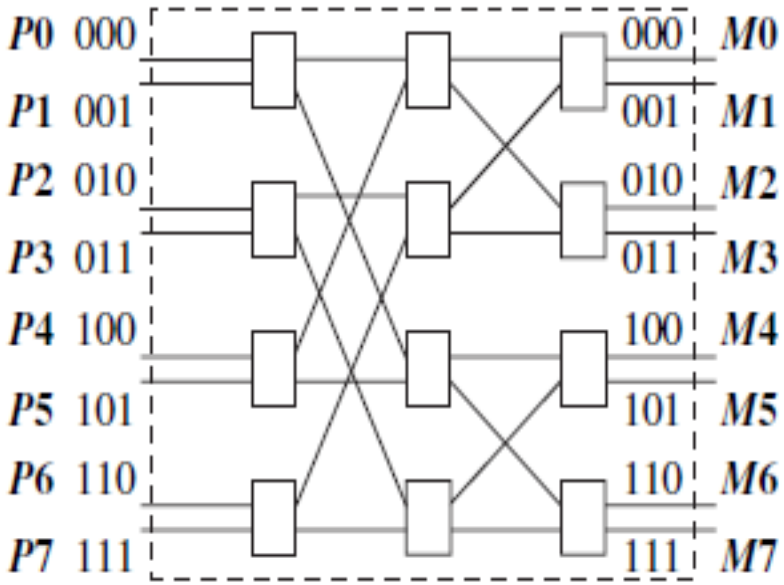
1.Multiprocessor system -Butterfly network

Consider the Butterfly network n = 8 and M = 4s

There are three stages s = 0, 1, 2

Interconnection pattern is defined between s = 0 and s = 1 and between s = 1 and s = 2.

The switch number x varies from 0 to 3 in each stage, i.e., x is a 2-bit string.



(b) 3-stage Butterfly network ($n = 8$, $M = 4$)

Relation to parallel multiprocessor/multicomputer systems

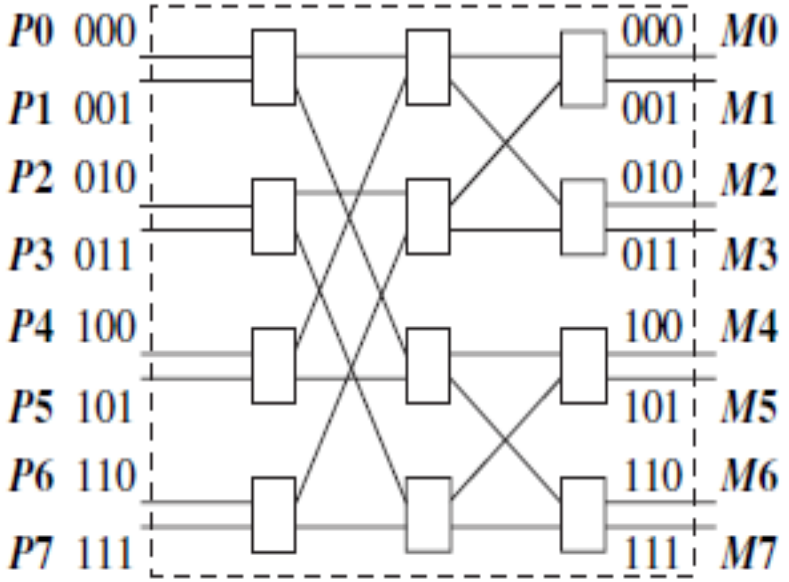
1.Multiprocessor system -Butterfly network

For stage s = 0, as per rule

(i)the first output line from switch 00 goes to the input line of switch 00 of stage s =1.

(ii)the second output line of switch 00 goes to input line of switch 10 of stage s = 1.

Similarly, x = 01 has one output line go to an input line of switch 11 in stage s = 1.



(b) 3-stage Butterfly network ($n = 8$, $M = 4$)

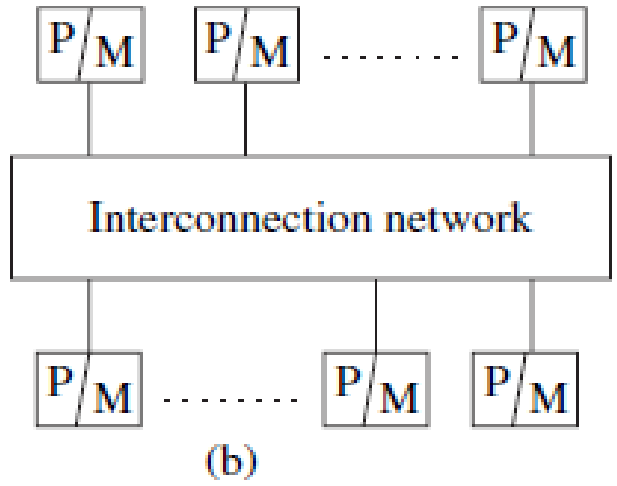
Relation to parallel multiprocessor/multicomputer systems

2.Multicomputer parallel system

A multicomputer parallel system is a parallel system in which the multiple processors do not have direct access to shared memory.

The memory of the multiple processors may or may not form a common address space.

Such computers usually do not have a common clock.



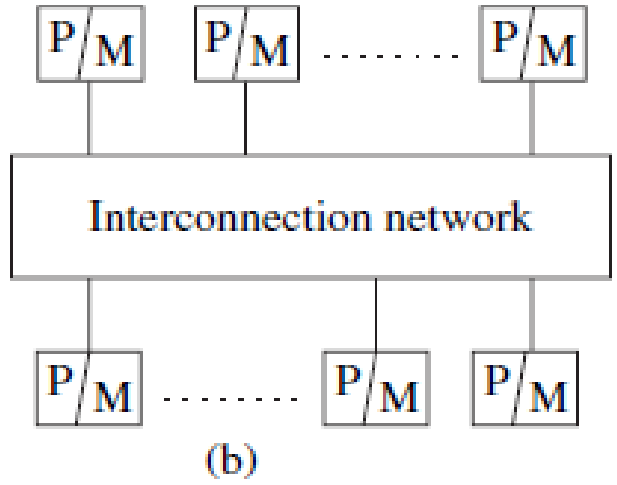
(b)

P processor
M memory

Relation to parallel multiprocessor/multicomputer systems

2.Multicomputer parallel system

- The processors communicate either via a common address space or via message-passing.
- A multicomputer system that has a common address space usually corresponds to a non-uniform memory access (NUMA) architecture



(b)

P processor
M memory

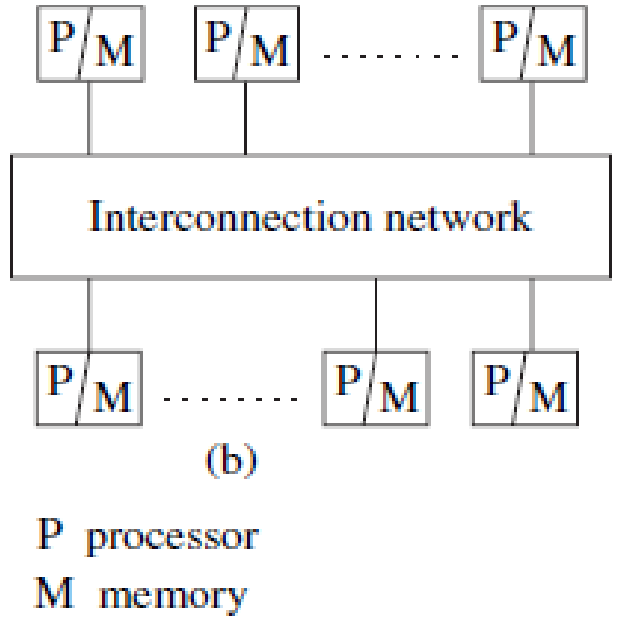
Relation to parallel multiprocessor/multicomputer systems

2.Multicomputer parallel system

Examples:

- NYU Ultracomputer
- Sequent shared memory machines
- CM* Connection machine and Processors configured in regular and symmetrical topologies such as an array or mesh, ring, torus, cube, and hypercube.

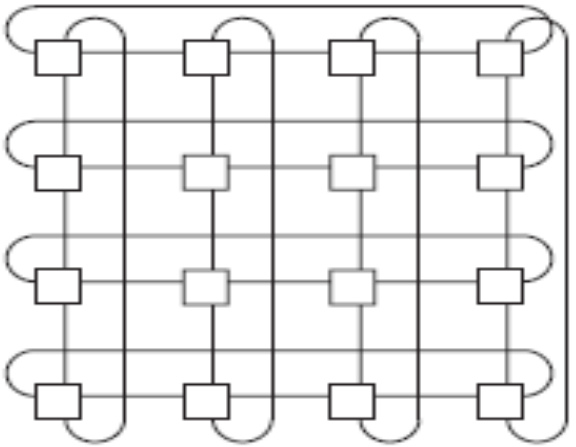
.



Relation to parallel multiprocessor/multicomputer systems

2.Multicomputer parallel system

- Wrap-around 4×4 mesh.
- For a k×k mesh which will contain k^2 processors, the maximum path length between any two processors is 2(k/2−1).
- Routing can be done along the Manhattan grid.



(a)

□ processor + memory

Figure 1.5 Some popular topologies for multicomputer shared-memory machines.
(a) Wrap-around 2D-mesh, also known as torus.

Relation to parallel multiprocessor/multicomputer systems

2.Multicomputer parallel system

- 4-dimensional hypercube
- A k-dimensional hypercube has 2^k processor-and-memory units.
- Each such unit is a node in the hypercube, and has a unique k-bit label.
- Each of the k dimensions is associated with a bit position in the label.

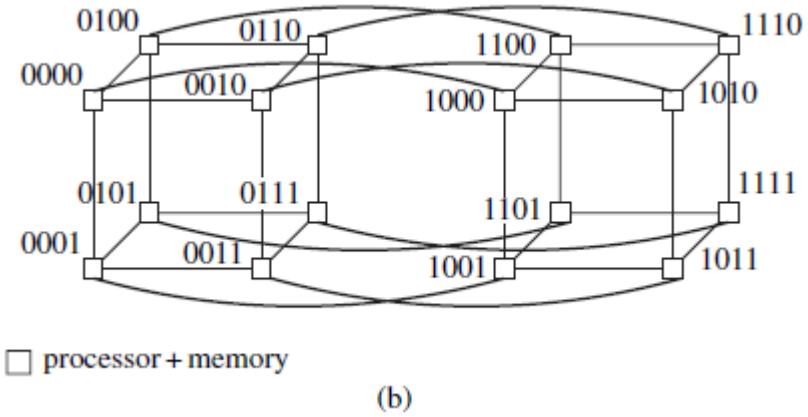


Figure 1.5 Some popular topologies for multicomputer shared-memory machines. (b) Hypercube of dimension 4.

Relation to parallel multiprocessor/multicomputer systems

2.Multicomputer parallel system

- The labels of any two adjacent nodes are identical except for the bit position corresponding to the dimension in which the two nodes differ.
- Processors are labelled such that the shortest path between any two processors is the Hamming distance between the processor labels.
- Nodes 0101 and 1100 have a Hamming distance of 2.
- The shortest path between them has length 2.

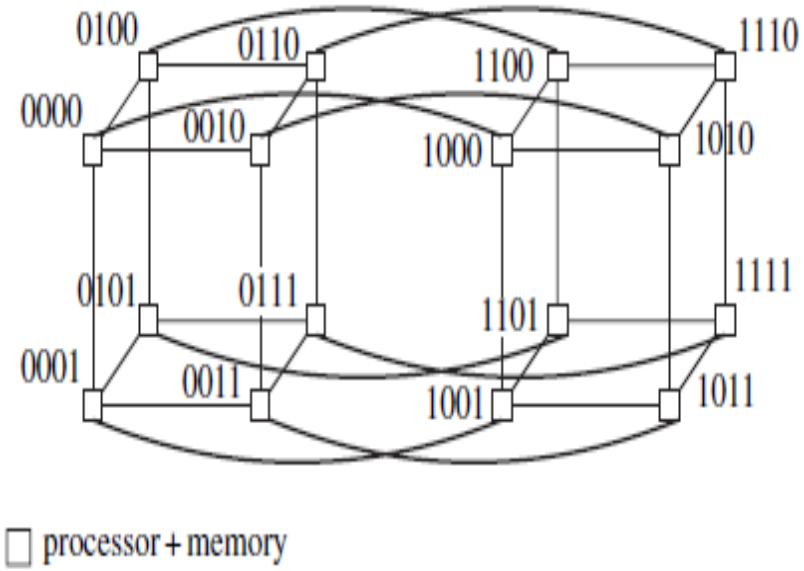


Figure 1.5 Some popular topologies for multicomputer shared-memory machines. (b) Hypercube of dimension 4.

Relation to parallel multiprocessor/multicomputer systems

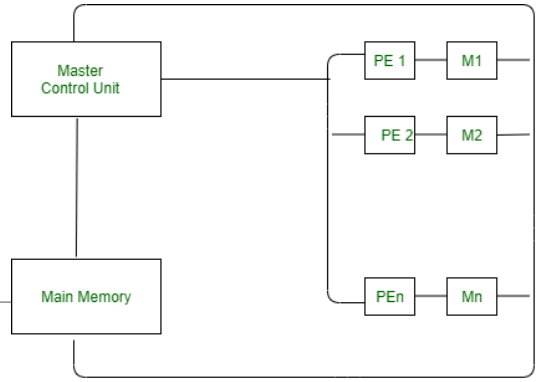


Figure - SIMD Array Processor Organization

3.Array processors

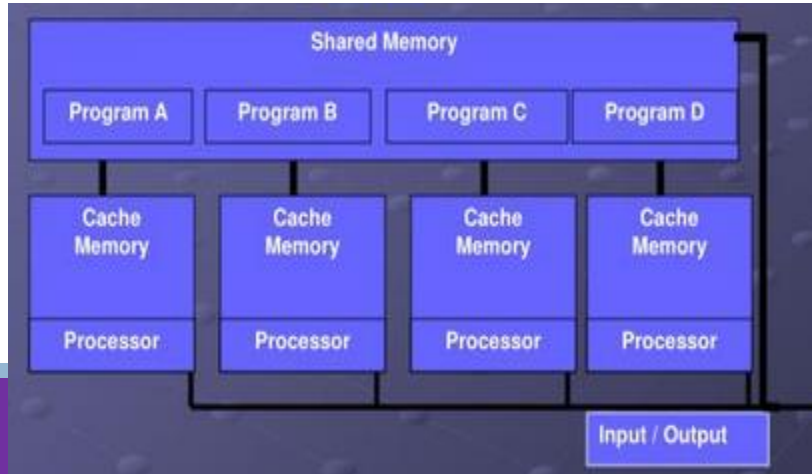
A class of parallel computers that are physically co-located, are very tightly coupled, and have a common system clock

Array processors and systolic arrays that perform tightly synchronized processing and data exchange in lock-step

These applications usually involve a large number of iterations on the data.

This class of parallel systems has a very niche market.

Example : Applications such as DSP and image processing



Relation to parallel multiprocessor/multicomputer systems

Flynn's taxonomy

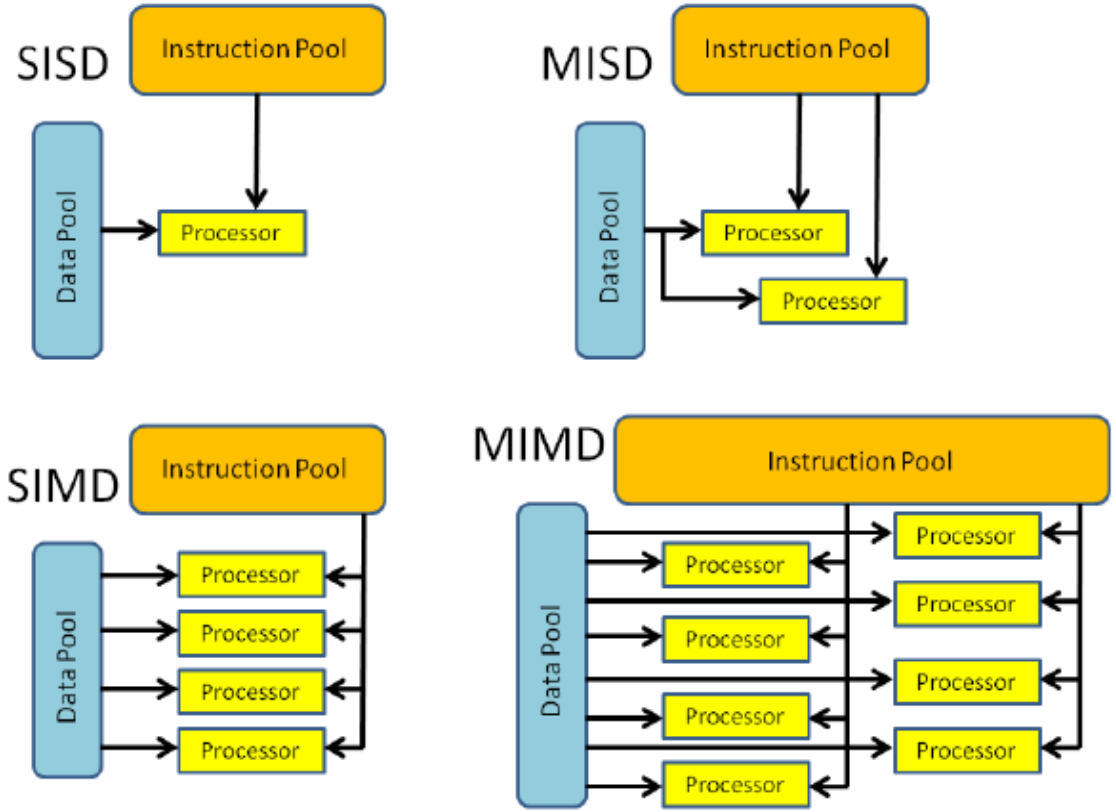
Flynn identified four processing modes, based on

- whether the processors execute the same or different instruction streams at the same time,
- Whether or not the processors processed the same (identical) data at the same time.

Relation to parallel multiprocessor/multicomputer systems

Flynn's taxonomy

1. Single instruction stream, single data stream (SISD)
2. Single instruction stream, multiple data stream (SIMD)
3. Multiple instruction stream, single data stream (MISD)
4. Multiple instruction stream, multiple data stream (MIMD)



Relation to parallel multiprocessor/multicomputer systems

Flynn's taxonomy

1. Single instruction stream, single data stream (SISD)

This mode corresponds to the conventional processing in the von Neumann paradigm with a single CPU, and a single memory unit connected by a system bus.

Relation to parallel multiprocessor/multicomputer systems

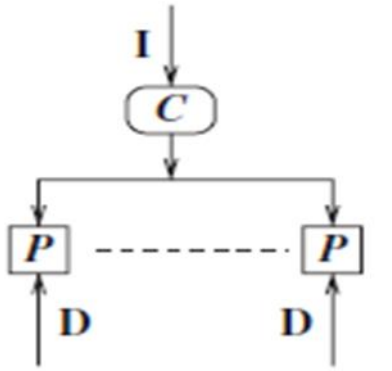
Flynn's taxonomy

2.Single instruction stream, multiple data stream (SIMD)

This mode corresponds to the processing by multiple homogenous processors which execute in lock-step on different data items.

Applications - large arrays and matrices

- scientific applications



(a) SIMD

C control unit
P processing unit
I instruction stream
D data stream

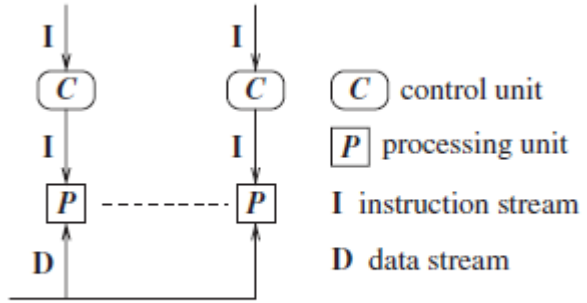
Relation to parallel multiprocessor/multicomputer systems

Flynn's taxonomy

3. Multiple instruction stream, single data stream (MISD)

This mode corresponds to the execution of different operations in parallel on the same data.

This is a specialized mode of operation with limited but niche applications, e.g., visualization.



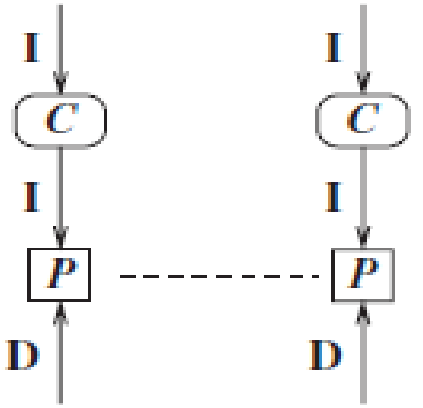
(c) MISD

Relation to parallel multiprocessor/multicomputer systems

Flynn's taxonomy

4. Multiple instruction stream, multiple data stream (MIMD)

- Various processors execute different code on different data.
- There is no common clock among the system processors.
- Examples :Sun Ultra servers, multicomputer PCs and IBM SP



(b) MIMD

Relation to parallel multiprocessor/multicomputer systems

Coupling

- The degree of coupling among a set of modules, whether hardware or software, is measured in terms of the interdependency and binding and/or homogeneity among the modules.
- When the degree of coupling is high (low), the modules are said to be tightly (loosely) coupled.
- SIMD and MISD architectures generally tend to be tightly coupled because of the common clocking of the shared instruction stream or the shared data stream.

Relation to parallel multiprocessor/multicomputer systems

Parallelism or speedup of a program on a specific system

- This is a measure of the relative speedup of a specific program, on a given machine.
- The speedup depends on the number of processors and the mapping of the code to the processors.
- It is expressed as the ratio of the time T1 with a single processor, to the time T(n) with n processors.

Relation to parallel multiprocessor/multicomputer systems

Parallelism within a parallel/distributed program

This is an aggregate measure of the percentage of time that all the processors are executing CPU instructions productively, as opposed to waiting for communication (either via shared memory or message-passing) operations to complete.

The term is traditionally used to characterize parallel programs.

Relation to parallel multiprocessor/multicomputer systems

Concurrency of a program

The parallelism/concurrency in a parallel/distributed program can be measured by the ratio of

- the number of local (non-communication and non-shared memory access) operations to the total number of operations, including the communication or shared memory access operations.

Relation to parallel multiprocessor/multicomputer systems

Granularity of a program

The ratio of the amount of computation to the amount of communication within the parallel/distributed program is termed as granularity

Relation to parallel multiprocessor/multicomputer systems

Granularity of a program

The ratio of the amount of computation to the amount of communication within the parallel/distributed program is termed as granularity

RAMAIAH Institute of Technology

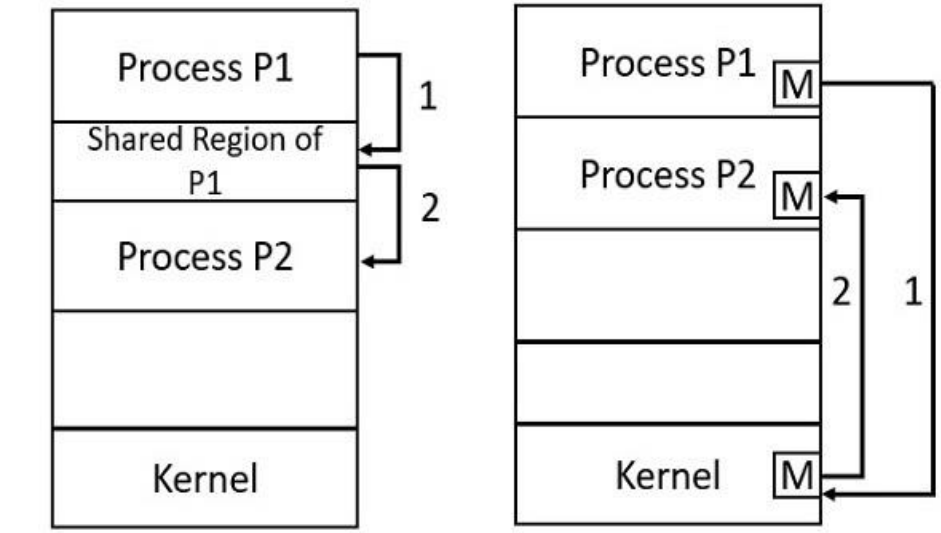
UNIT -1

Introduction:(Chapter 1-1.1,1.2,1.3,1.4,1.5,1.6,1.7,1.8)

- Definition
- Relation to computer system components
- Motivation
- Relation to parallel multiprocessor/multicomputer systems,
- Message-passing systems versus shared memory systems.
- Primitives for distributed communication
- Synchronous versus asynchronous executions
- Design issues and challenges

Message-passing systems versus shared memory systems

- Shared memory systems are those in which there is a (common) shared address space throughout the system.
- Communication among processors takes place via shared data variables, and control variables for synchronization among the processors.
- Semaphores and monitors that were originally designed for shared memory uniprocessors and multiprocessors are examples of how synchronization can be achieved in shared memory systems.



Shared Memory System Message Passing System

Message-passing systems versus shared memory systems

- **Emulating message-passing on a shared memory system (**MP →SM**)**
- The shared address space can be partitioned into disjoint parts, one part being assigned to each processor.
- “Send” and “receive” operations can be implemented by writing to and reading from the destination/sender processor’s address space, respectively.
- Specifically, a separate location can be reserved as the mailbox for each ordered pair of processes.

Message-passing systems versus shared memory systems

- **Emulating message-passing on a shared memory system (**MP →SM**)**
- A Pi–Pj message-passing can be emulated by a write by Pi to the mailbox and then a read by Pj from the mailbox.
- In the simplest case, these mailboxes can be assumed to have unbounded size.
- The write and read operations need to be controlled using synchronization rimitives to inform the receiver/sender after the data has been sent/received.

Message-passing systems versus shared memory systems

- **Emulating shared memory on a message-passing system (**SM →MP**)**
- This involves the use of “send” and “receive” operations for “write” and “read” operations.
- Each shared location can be modeled as a separate process;
  - “write” to a shared location is emulated by sending an update message to the corresponding owner process;
  - “read” to a shared location is emulated by sending a query message to the owner process.

RAMAIAH Institute of Technology

UNIT -1

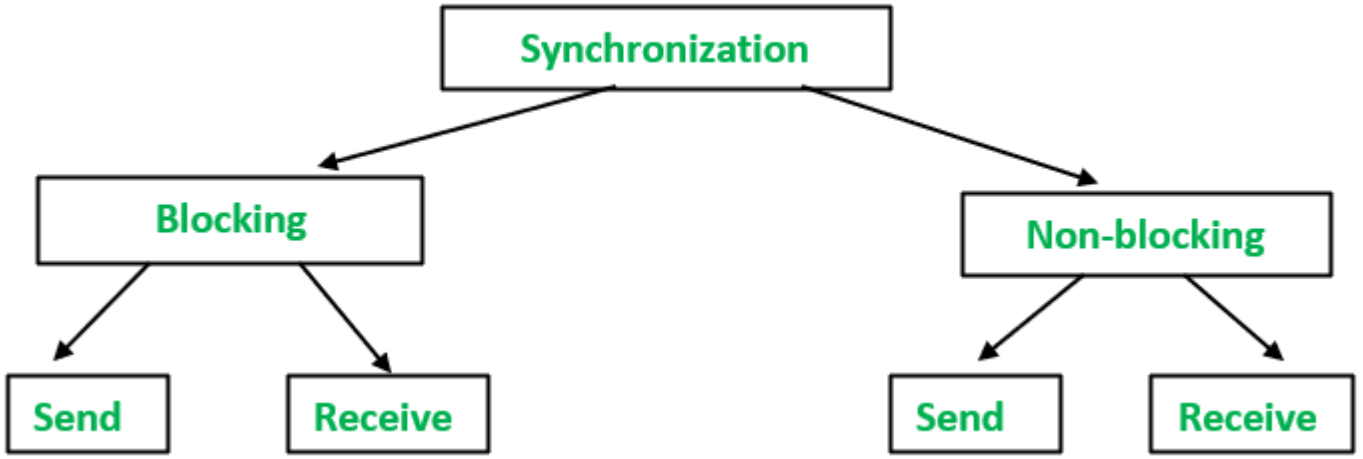
Introduction:(Chapter 1-1.1,1.2,1.3,1.4,1.5,1.6,1.7,1.8)

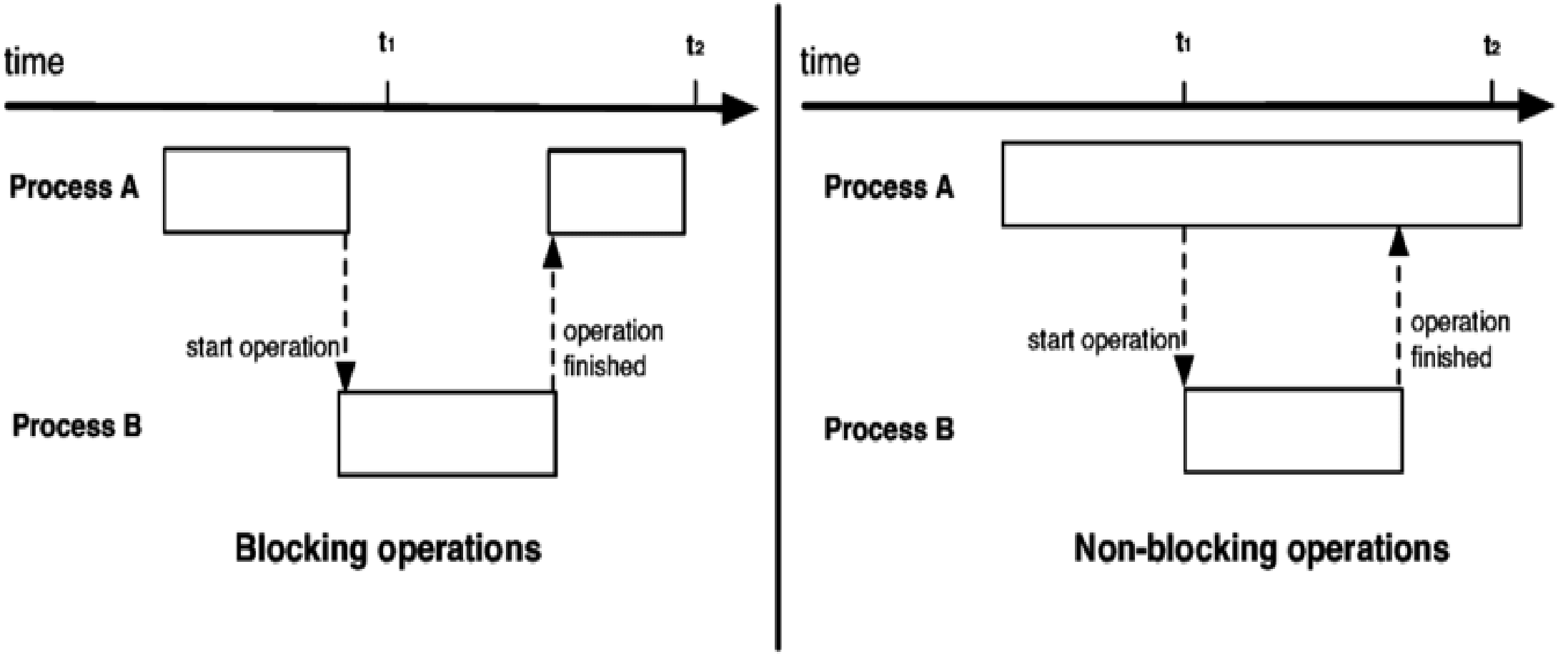
- Definition
- Relation to computer system components
- Motivation
- Relation to parallel multiprocessor/multicomputer systems,
- Message-passing systems versus shared memory systems.
- Primitives for distributed communication
- Synchronous versus asynchronous executions
- Design issues and challenges

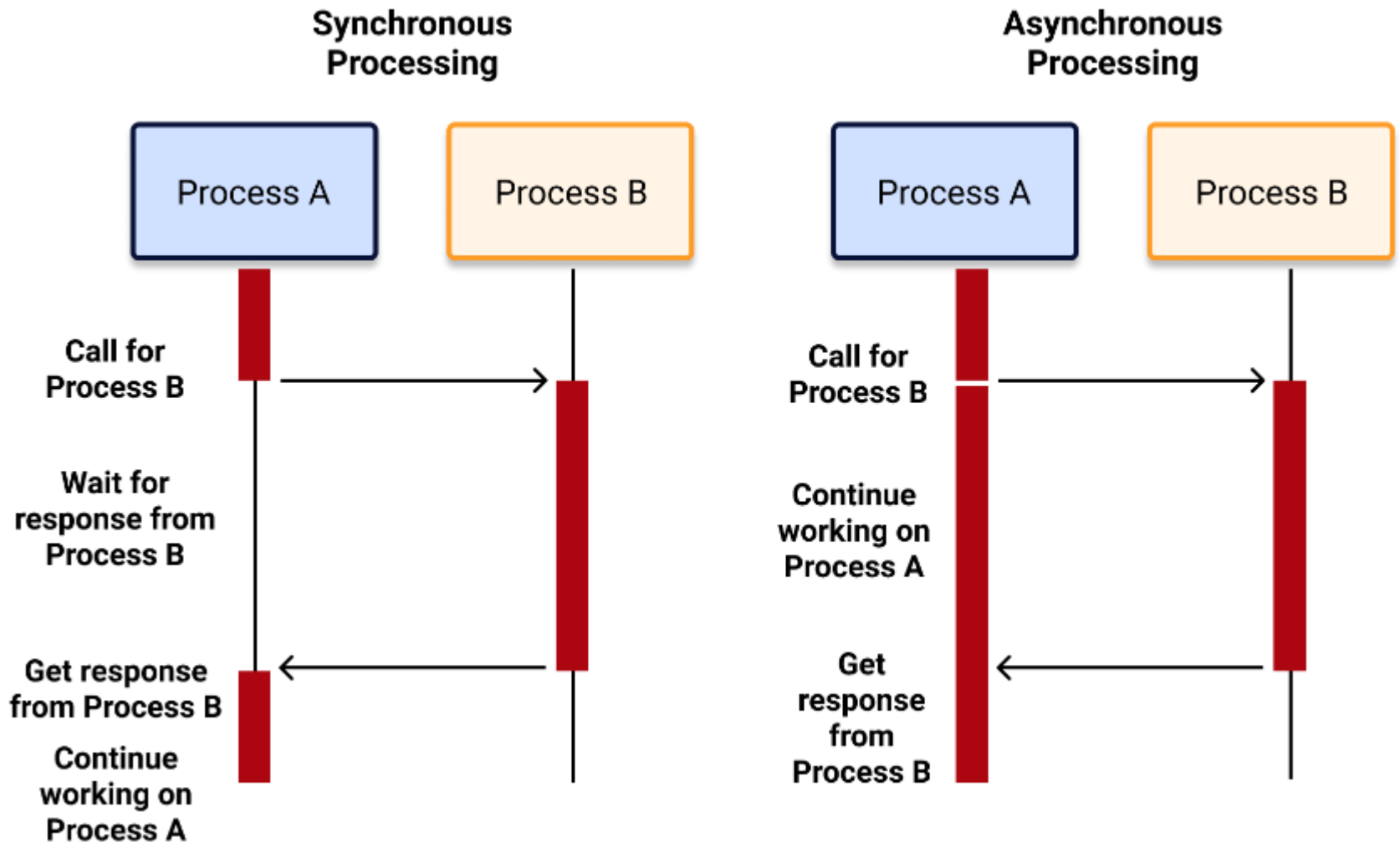
Primitives for distributed communication

Different types of Primitives

- Blocking/non-blocking
- Synchronous/Asynchronous primitives




time
t1
t2
Process A
start operation
operation
finished
Process B
Blocking operations
time
t1
t2
Process A
start operation
operation
finished
Process B
Non-blocking operations


Synchronous Processing
Process A
Process B
Call for Process B
Wait for response from Process B
Get response from Process B
Continue working on Process A
Asynchronous Processing
Process A
Process B
Call for Process B
Continue working on Process A
Get response from Process B

Primitives for distributed communication

Message send and message receive communication primitives are denoted Send() and Receive(), respectively.

- A Send primitive has at least two parameters

– the destination, and the buffer in the user space, containing the data to be sent.

- A Receive primitive has at least two parameters

- the source from which the data is to be received and the user buffer into which the data is to be received.

Primitives for distributed communication

There are two ways of sending data when the Send primitive is invoked –

- the buffered option
- the unbuffered option

The buffered option which is the standard option copies the data from the user buffer to the kernel buffer.

The data later gets copied from the kernel buffer onto the network.

Primitives for distributed communication

The Unbuffered option, the data gets copied directly from the user buffer onto the network.

For the Receive primitive, the buffered option is usually required because the data may already have arrived when the primitive is invoked, and needs a storage place in the kernel.

Primitives for distributed communication

- Synchronous
  - The sender is blocked until its message is stored in the local **buffer** at the receiving host or delivered to the **receiver**.
- Asynchronous
  - The sender continues immediately after executing a send
  - The message is stored in the local buffer at the **sending host** or at the **first communication server**.

Primitives for distributed communication

- Synchronous (send/receive)
  - Handshake between sender and receiver
  - Send completes when Receive completes
  - Receive completes when data copied into buffer
- Asynchronous (send)
  - Control returns to process when data copied out of user-specified buffer

Primitives for distributed communication

Synchronous primitives

- A Send or a Receive primitive is synchronous if both the Send() and Receive() handshake with each other.
  - The processing for the Send primitive completes only after the invoking processor learns that the other corresponding Receive primitive has also been invoked and that the receive operation has been completed.
  - The processing for the Receive primitive completes when the data to be received is copied into the receiver's user buffer.

Primitives for distributed communication

Asynchronous primitives

A Send primitive is said to be asynchronous if control returns back to the invoking process after the data item to be sent has been copied out of the user-specified buffer.

It does not make sense to define asynchronous Receive primitives.

Primitives for distributed communication

- Blocking (send/receive)

  - Control returns to invoking process after processing of primitive (whether sync or async) completes

- Nonblocking (send/receive)

  - Control returns to process immediately after invocation

  - Send: even before data copied out of user buffer

  - Receive: even before data may have arrived from sender

Primitives for distributed communication

Blocking primitives

A primitive is blocking if control returns to the invoking process after the processing for the primitive (whether in synchronous or asynchronous mode) completes.

Primitives for distributed communication

Non-blocking primitives

A primitive is non-blocking if control returns back to the invoking process immediately after invocation, even though the operation has not completed.

For a non-blocking Send, control returns to the process even before the data is copied out of the user buffer.

For a non-blocking Receive, control returns to the process even before the data may have arrived from the sender.

Primitives for distributed communication

Non-blocking Send primitive

A return parameter on the primitive call returns a system-generated handle

- which can be later used to check the status of completion of the call.

The Wait call usually blocks until one of the parameter handles is posted.

```
Send(X, destination, handle_k)                    // handle_k is a return parameter
...
...
Wait(handle_1, handle_2, ..., handle_k, ..., handle_m)    // Wait always blocks
```

Primitives for distributed communication

4 versions of the Send primitive

1. synchronous blocking
2. synchronous non-blocking
3. asynchronous blocking
4. Asynchronous non-blocking

These versions of the primitives are Illustrated using a timing diagram.

2 versions of the Receive primitive

1. blocking synchronous
2. non-blocking synchronous

Primitives for distributed communication

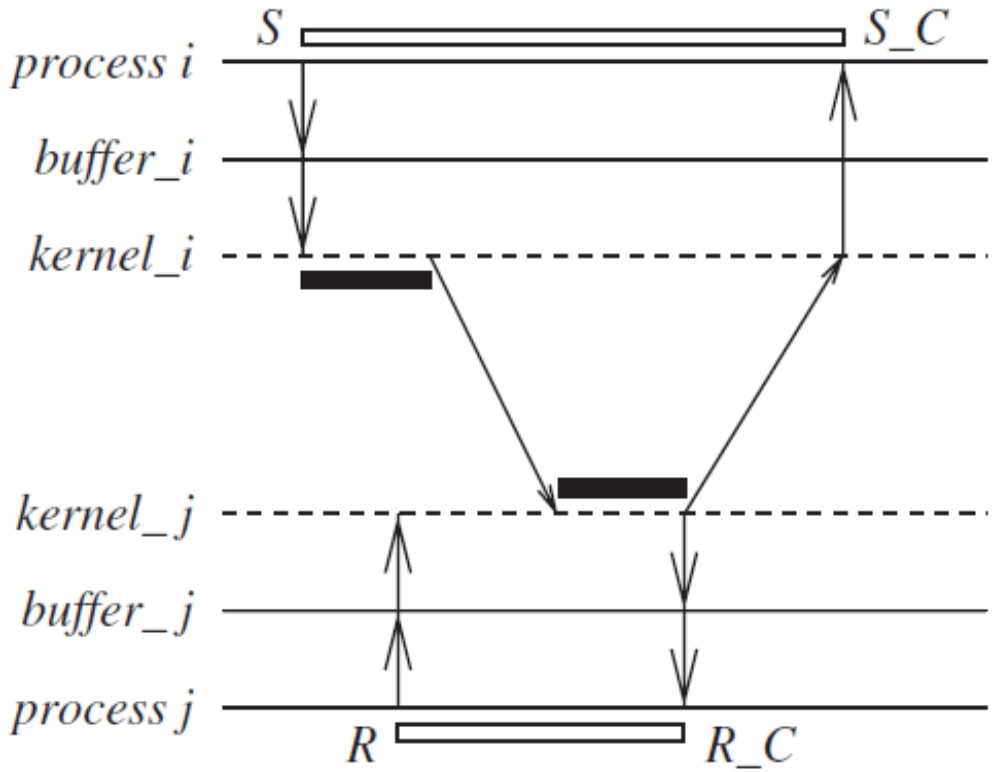
Process Pi is sending and process Pj is receiving

3 time lines are shown for each process:

1. for the process execution

2. for the user buffer from/to which data is sent/received

3. for the kernel/communication subsystem.

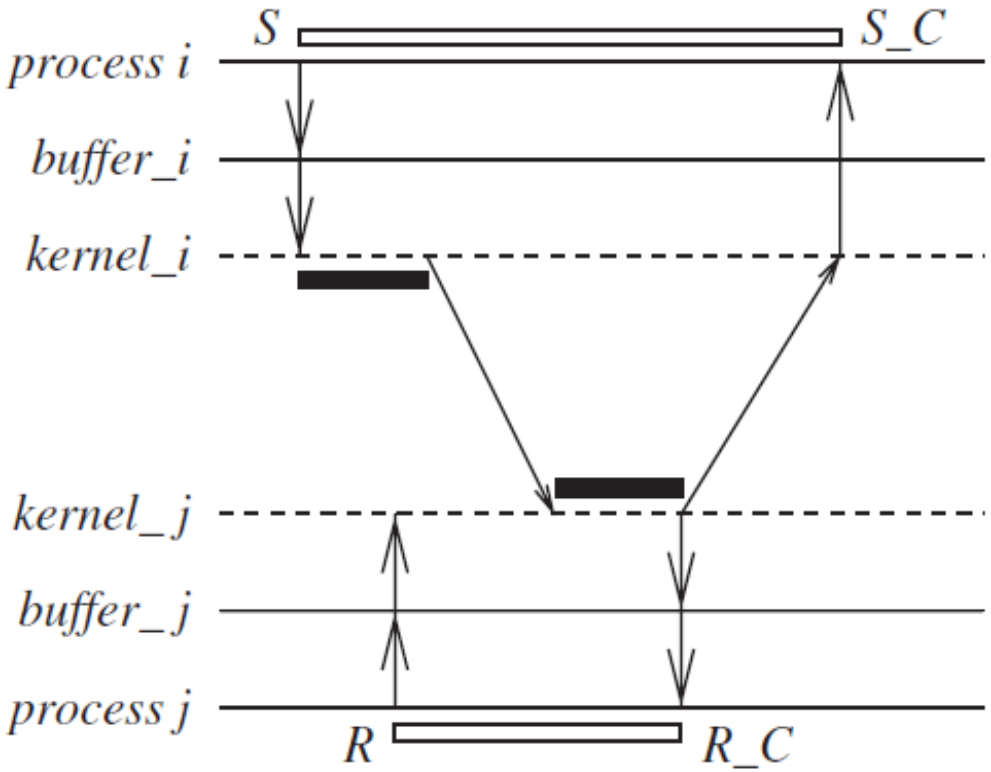


(a) Blocking sync. *Send*, blocking *Receive*

Primitives for distributed communication

Process Pi is sending and process Pj is receiving

▬	Duration to copy data from or to user buffer
▭	Duration in which the process issuing send or receive primitive is blocked
S	*Send* primitive issued
R	*Receive* primitive issued
P	The completion of the previously initiated nonblocking operation
W	Process may issue *Wait* to check completion of nonblocking operation
S_C	processing for *Send* completes
R_C	processing for *Receive* completes

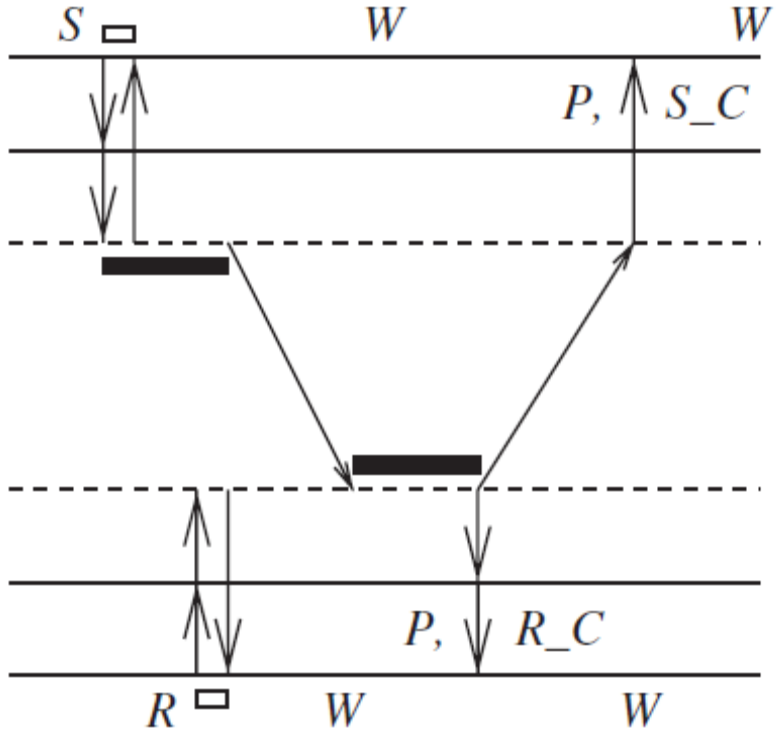


(a) Blocking sync. *Send*, blocking *Receive*

Primitives for distributed communication

Process Pi is sending and process Pj is receiving

▬	Duration to copy data from or to user buffer
▭	Duration in which the process issuing send or receive primitive is blocked
S	*Send* primitive issued
R	*Receive* primitive issued
P	The completion of the previously initiated nonblocking operation
W	Process may issue *Wait* to check completion of nonblocking operation
S_C	processing for *Send* completes
R_C	processing for *Receive* completes

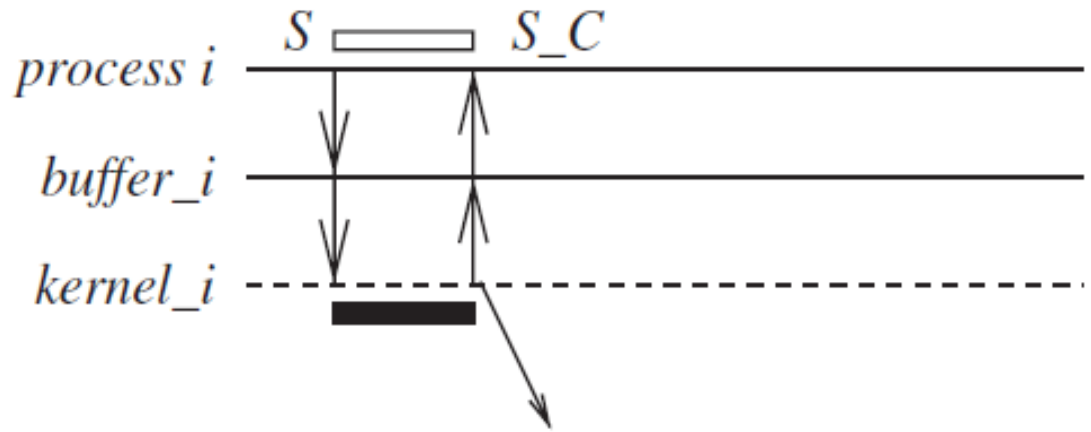


(b) Nonblocking sync. *Send*, nonblocking *Receive*

Primitives for distributed communication

Process Pi is sending and process Pj is receiving

▬	Duration to copy data from or to user buffer
▭	Duration in which the process issuing send or receive primitive is blocked
S	*Send* primitive issued
R	*Receive* primitive issued
P	The completion of the previously initiated nonblocking operation
W	Process may issue *Wait* to check completion of nonblocking operation
S_C	processing for *Send* completes
R_C	processing for *Receive* completes



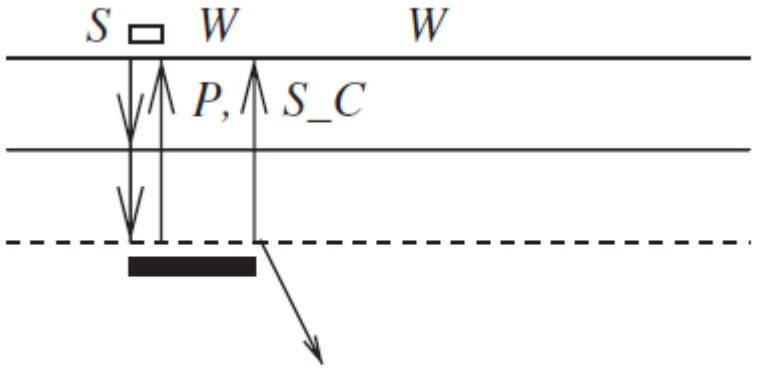
(c) Blocking async. *Send*

RAMAIAH Institute of Technology

Primitives for distributed communication

Process Pi is sending and process Pj is receiving

▬	Duration to copy data from or to user buffer
▭	Duration in which the process issuing send or receive primitive is blocked
S	*Send* primitive issued
R	*Receive* primitive issued
P	The completion of the previously initiated nonblocking operation
W	Process may issue *Wait* to check completion of nonblocking operation
S_C	processing for *Send* completes
R_C	processing for *Receive* completes



(d) Non-blocking async. *Send*

Primitives for distributed communication

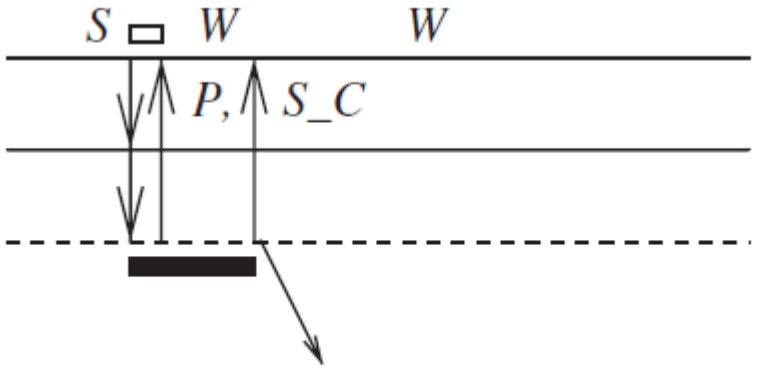
Processor synchrony

1. Processor synchrony indicates that all the processors execute in lock-step with their clocks synchronized.
2. As this synchrony is not attainable in a distributed system, what is more generally indicated is that for a large granularity of code, usually termed as a step, the processors are synchronized.
3. This abstraction is implemented using some form of barrier synchronization to ensure that no processor begins executing the next step of code until all the processors have completed executing the previous steps of code assigned to each of the processors.

Primitives for distributed communication

Process Pi is sending and process Pj is receiving

▬	Duration to copy data from or to user buffer
▭	Duration in which the process issuing send or receive primitive is blocked
S	*Send* primitive issued
R	*Receive* primitive issued
P	The completion of the previously initiated nonblocking operation
W	Process may issue *Wait* to check completion of nonblocking operation
S_C	processing for *Send* completes
R_C	processing for *Receive* completes



(d) Non-blocking async. *Send*

RAMAIAH Institute of Technology

UNIT -1

Introduction:(Chapter 1-1.1,1.2,1.3,1.4,1.5,1.6,1.7,1.8)

- Definition
- Relation to computer system components
- Motivation
- Relation to parallel multiprocessor/multicomputer systems,
- Message-passing systems versus shared memory systems.
- Primitives for distributed communication
- Synchronous versus asynchronous executions
- Design issues and challenges

Synchronous versus asynchronous executions

Another classification namely

1. Synchronous executions
2. Asynchronous executions

Synchronous versus asynchronous executions

Asynchronous executions

An asynchronous execution is an execution in which

(i) there is no processor synchrony and there is no bound on the drift rate of processor clocks

(ii) message delays (transmission + propagation times) are finite but unbounded

(iii) there is no upper bound on the time taken by a process to execute a step.

RAMAIAH Institute of Technology

Synchronous versus asynchronous executions

An example asynchronous execution with four processes P0 to P3 is shown in Figure 1.9.

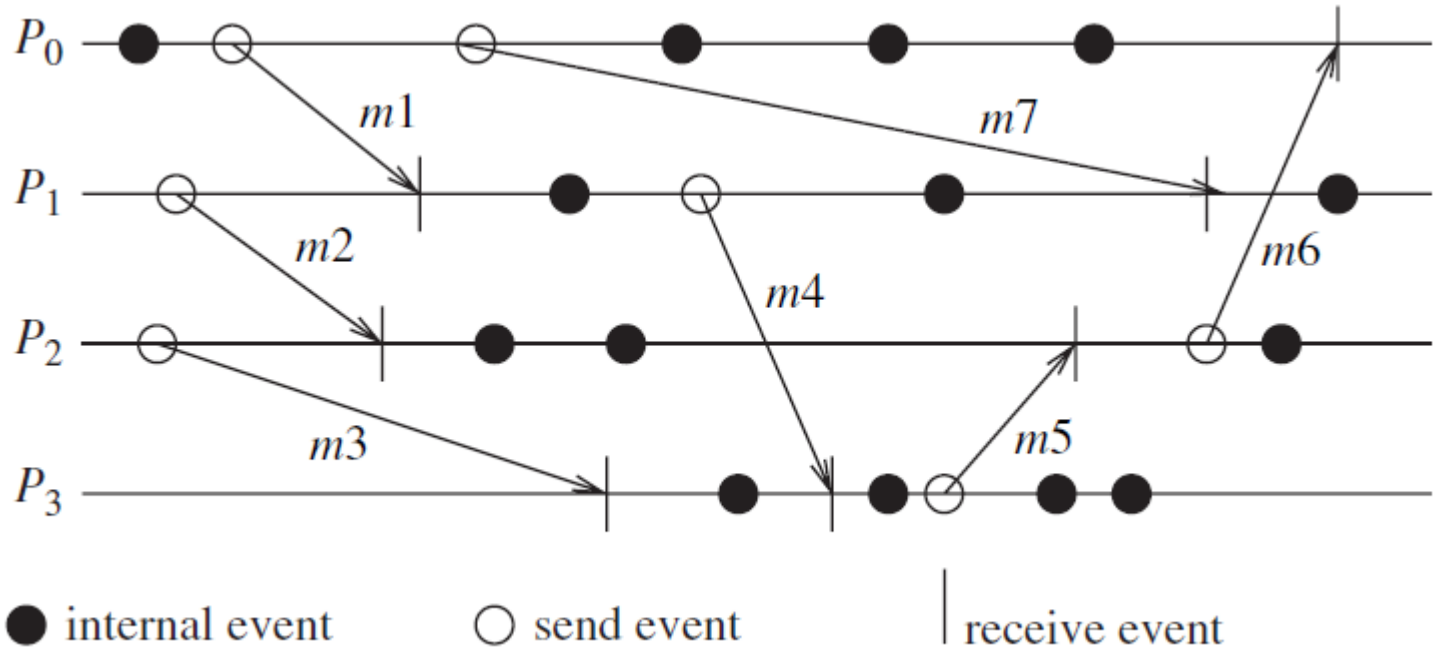


Figure 1.9 An example of an asynchronous execution in a message-passing system. A timing diagram is used to illustrate the execution.

RAMAIAH
Institute of Technology

Synchronous versus asynchronous executions

- The arrows denote the messages; tail –send event denoted by a circle

  head - receive event denoted by vertical line

- Non-communication events, also termed as internal events, are shown by shaded circles.

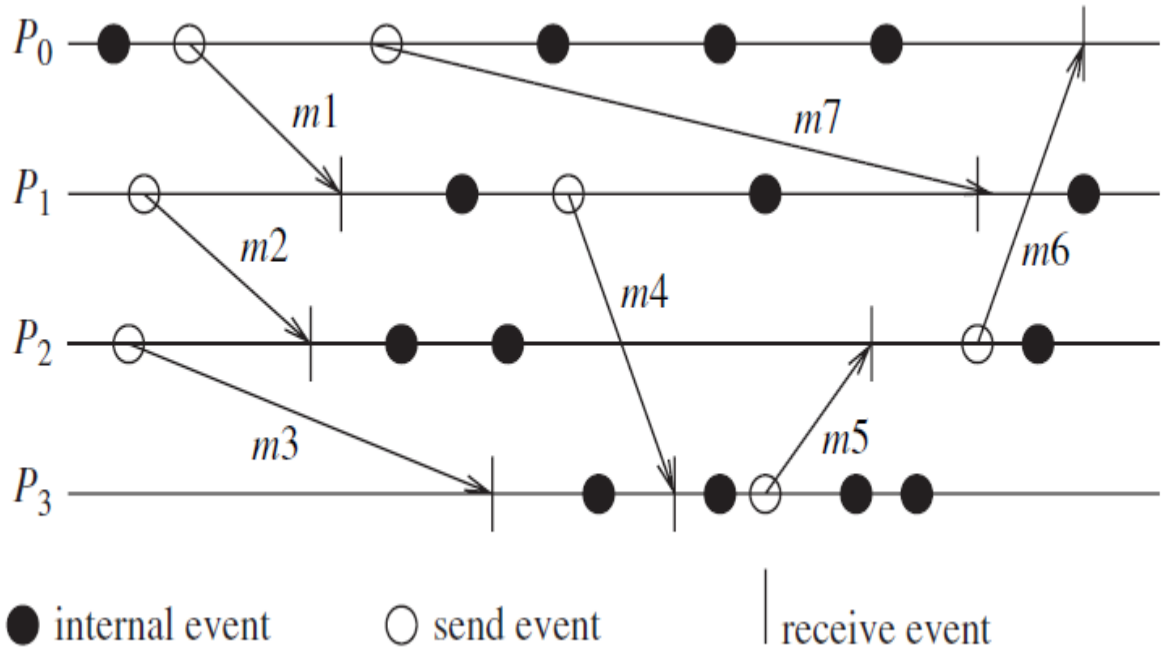


Figure 1.9 An example of an asynchronous execution in a message-passing system. A timing diagram is used to illustrate the execution.

RAMAIAH Institute of Technology

Synchronous versus asynchronous executions

- The arrows denote the messages; tail –send event denoted by a circle

  head - receive event denoted by vertical line

- Non-communication events, also termed as internal events, are shown by shaded circles.

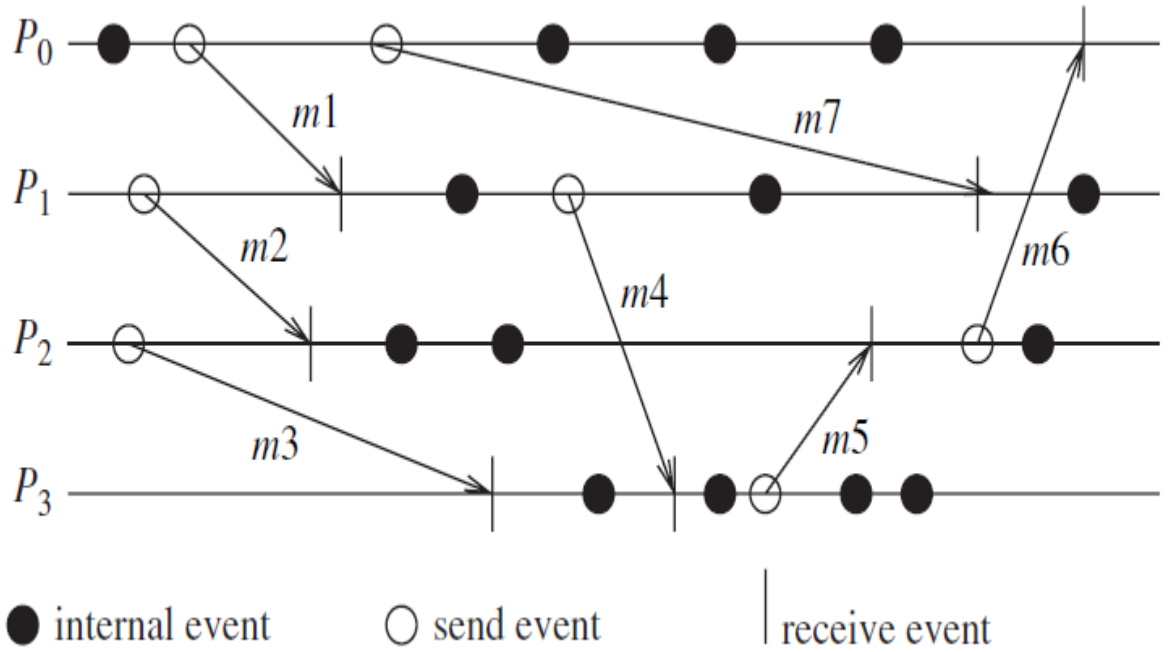


Figure 1.9 An example of an asynchronous execution in a message-passing system. A timing diagram is used to illustrate the execution.

Synchronous versus asynchronous executions

Synchronous executions

A synchronous execution is an execution in which

(i) processors are synchronized and the clock drift rate between any two processors is bounded,

(ii) message delivery (transmission + delivery) times are such that they occur in one logical step or round,

(iii) there is a known upper bound on the time taken by a process to execute a step.

Synchronous versus asynchronous executions

An example of a synchronous execution with four processes P0 to P3 is shown in Figure 1.10.

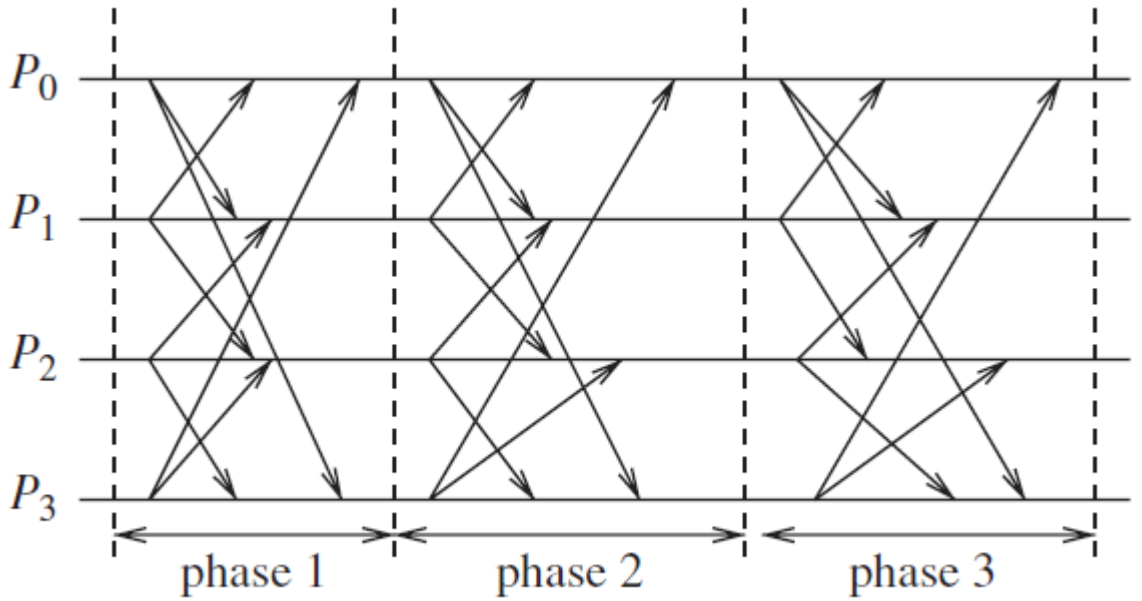


Figure 1.10 An example of a synchronous execution in a message-passing system. All the messages sent in a round are received within that same round.

RAMAIAH Institute of Technology

Synchronous versus asynchronous executions

In each round, process Pi sends a message to $P_{(i+1) \bmod 4}$ and $P_{(i-1) \bmod 4}$ and calculates some application-specific function on the received values.

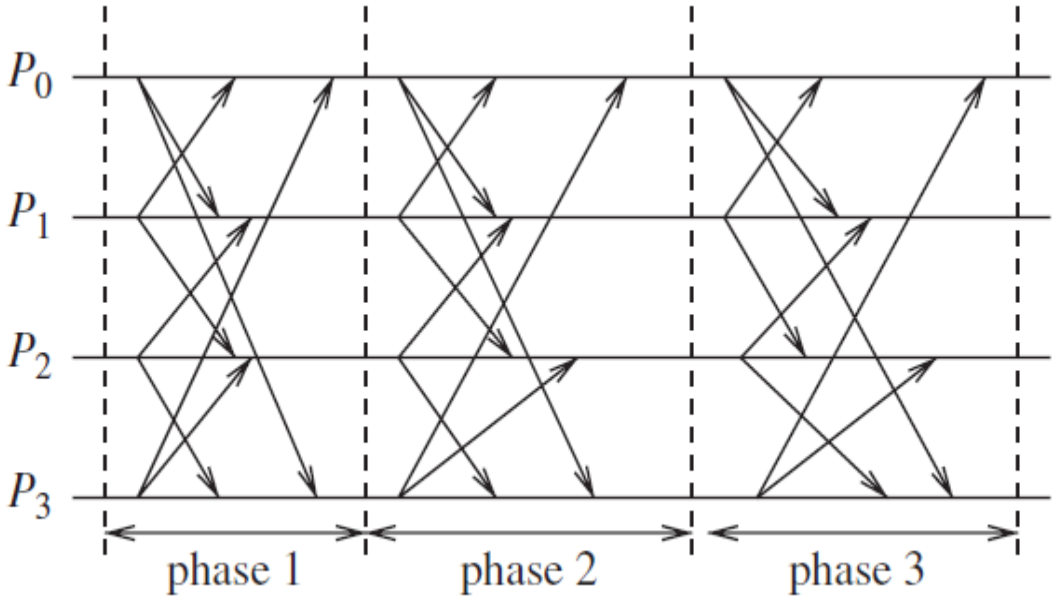


Figure 1.10 An example of a synchronous execution in a message-passing system. All the messages sent in a round are received within that same round.

Synchronous versus asynchronous executions

- Difficult to build a completely synchronous system, and have the messages delivered within a bounded time.
- Virtual Synchrony
  - Asynchronous execution, processes synchronize as per application requirement
  - Execute in round/steps
- Emulation

Synchronous versus asynchronous executions

Emulating an asynchronous system by a synchronous system (A→S)

An asynchronous program (written for an asynchronous system) can be emulated on a synchronous system fairly trivially

- as the synchronous system is a special case of an asynchronous system – all communication finishes within the same round in which it is initiated.

Synchronous versus asynchronous executions

Emulating a synchronous system by an asynchronous system (S →A)

A synchronous program (written for a synchronous system) can be emulated on an asynchronous system

- using a tool called synchronizer

Synchronous versus asynchronous executions

Emulations

4 broad classes of programs, as shown in Figure 1.11.

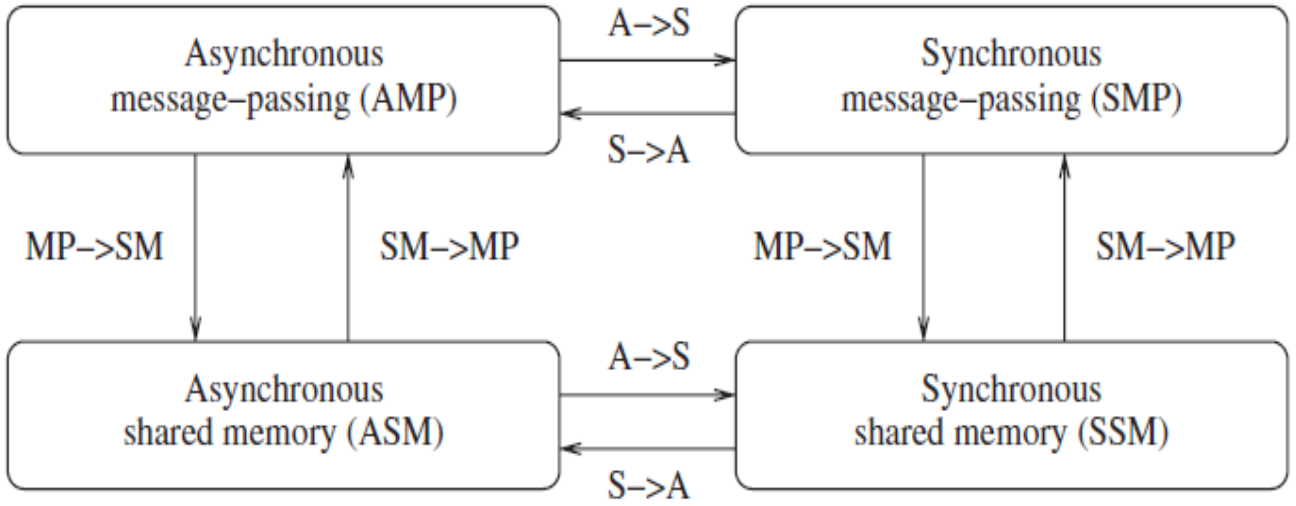


Figure 1.11 Emulations among the principal system classes in a failure-free system.

Synchronous versus asynchronous executions

Emulations

Any class can be emulated by any other.

If system A can be emulated by system B, denoted A/B, and if a problem is not solvable in B, then it is also not solvable in A.

All four classes are equivalent in terms of computability" – what can and cannot be computed – in failure-free systems.

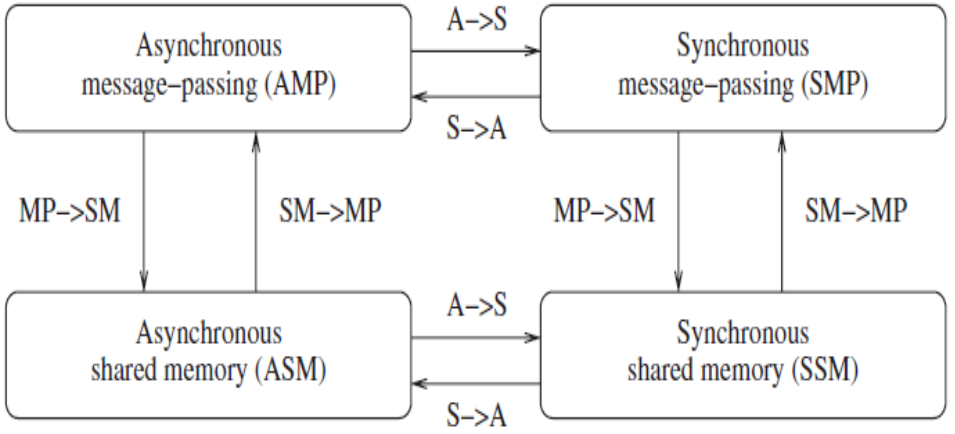


Figure 1.11 Emulations among the principal system classes in a failure-free system.

RAMAIAH Institute of Technology

UNIT -1

Introduction:(Chapter 1-1.1,1.2,1.3,1.4,1.5,1.6,1.7,1.8)

- Definition
- Relation to computer system components
- Motivation
- Relation to parallel multiprocessor/multicomputer systems,
- Message-passing systems versus shared memory systems.
- Primitives for distributed communication
- Synchronous versus asynchronous executions
- Design issues and challenges

Design issues and challenges

- Distributed systems challenges from a system perspective
- Algorithmic challenges in distributed computing
- Applications of distributed computing and newer challenges

RAMAIAH Institute of Technology

Design issues and challenges

Distributed systems challenges from a system perspective

1. Communication
2. Processes
3. Naming
4. Synchronization
5. Data storage and access
6. Consistency and replication
7. Fault tolerance
8. Security
9. Applications Programming Interface (API) and transparency
10. Scalability and modularity

Main challenges in designing distributed systems from a system building perspective.

RAMAIAH Institute of Technology

Design issues and challenges

Distributed systems challenges from a system perspective

Communication - Remote procedure call (RPC), Remote object invocation (ROI), message-oriented communication versus stream-oriented communication.

Processes -Some of the issues involved are: management of processes and threads at clients/servers; code migration; and the design of software and mobile agents.

Naming - Naming in mobile systems provides additional challenges because naming cannot easily be tied to any static geographical topology.

Synchronization-Mutual exclusion, leader election, synchronizing physical clocks and devising logical clocks that capture the essence of the passage of time, as well as global state recording algorithms

RAMAIAH Institute of Technology

Design issues and challenges

Distributed systems challenges from a system perspective

Data storage and access - Traditional issues such as file system design have to be reconsidered in the setting of a distributed system.

Consistency and replication- Issues of managing the replicas, and dealing with consistency among the replicas/caches in a distributed setting.

Fault tolerance -Fault tolerance requires maintaining correct and efficient operation in spite of any failures of links, nodes, and processes.

Process resilience, reliable communication, distributed commit, checkpointing and recovery, agreement and consensus, failure detection, and self-stabilization are some of the mechanisms to provide fault-tolerance.

RAMAIAH Institute of Technology

Design issues and challenges

Distributed systems challenges from a system perspective

Security- various aspects of cryptography, secure channels, access control, key management – generation and distribution, authorization, and secure group management.

Scalability and modularity-The algorithms, data (objects), and services must be as distributed as possible.

Various techniques such as replication, caching and cache management, and asynchronous processing help to achieve scalability.

Design issues and challenges

Distributed systems challenges from a system perspective

Applications Programming Interface (API) and transparency-

The API for communication - ease of use and wider adoption of the distributed systems services by non-technical users.

Design issues and challenges

Distributed systems challenges from a system perspective

Applications Programming Interface (API) and transparency-

- Transparency deals with hiding the implementation policies from the user.

Transparency	Description
Access	Hide differences in data representation and how a resource is accessed
Location	Hide where a resource is located
Migration	Hide that a resource may move to another location
Relocation	Hide that a resource may be moved to another location while in use
Replication	Hide that a resource may be replicated
Concurrency	Hide that a resource may be shared by several competitive users
Failure	Hide the failure and recovery of a resource
Persistence	Hide whether a (software) resource is in memory or on disk

RAMAIAH Institute of Technology

Design issues and challenges

Distributed systems challenges from a system perspective

Transparency can be classified

1. Access transparency hides differences in data representation on different systems and provides uniform operations to access system resources.
2. Location transparency makes the locations of resources transparent to the users.
3. Migration transparency allows relocating resources without changing names. The ability to relocate the resources as they are being accessed is relocation transparency.

RAMAIAH Institute of Technology

Design issues and challenges

Distributed systems challenges from a system perspective

Transparency Can be classified

4.Replication transparency does not let the user become aware of any replication.

5.Concurrency transparency deals with masking the concurrent use of shared resources for the user.

6.Failure transparency refers to the system being reliable and fault-tolerant.

Design issues and challenges

Algorithmic challenges in distributed computing

1. Designing useful execution models and frameworks
2. Dynamic distributed graph algorithms and distributed routing algorithms
3. Time and global state in a distributed system
4. Synchronization/coordination mechanisms
5. Group communication, multicast, and ordered message delivery
6. Monitoring distributed events and predicates
7. Distributed program design and verification tools
8. Debugging distributed programs
9. Data replication, consistency models, and caching

load balancing
real time scheduling

Design issues and challenges

1.Designing useful execution models and frameworks

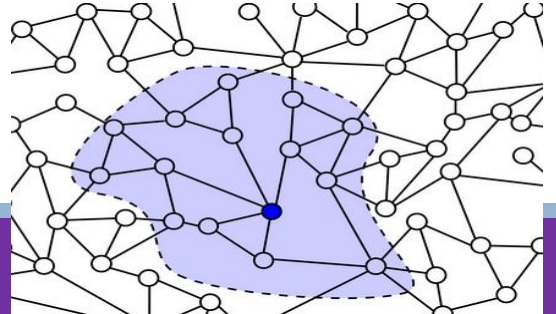
- Two widely adopted models of distributed system executions, useful for operational reasoning and the design of distributed algorithms.
  - The interleaving model
  - Partial order model

- To provide different degrees of infrastructure for reasoning and proving the correctness of distributed programs.
  - Input/output automata model
  - TLA (temporal logic of actions)

Design issues and challenges

2. Dynamic distributed graph algorithms and distributed routing algorithms

- The distributed system is modeled as a distributed graph
- The algorithms need to deal with dynamically changing graph characteristics, such as to model varying link loads in a routing algorithm.
- The efficiency of these algorithms impacts not only the user-perceived latency but also the traffic and hence the load or congestion in the network.
- Hence, the design of efficient distributed graph algorithms is of paramount importance.



Design issues and challenges

3. Time and global state in a distributed system

The challenges pertain to providing accurate physical time, and to providing a variant of time, called logical time.

Design issues and challenges

4.Synchronization/coordination mechanisms

- The synchronization mechanisms can also be viewed as resource management and concurrency management mechanisms to streamline the behavior of the processes that would otherwise act independently
- Synchronization is essential for the distributed processes to overcome the limited observation of the system state from the viewpoint of any one process.
- Overcoming this limited observation is necessary for taking any actions that would impact other processes.

RAMAIAH Institute of Technology

Design issues and challenges

4.Synchronization/coordination mechanisms

Here are some examples of problems requiring synchronization:

- Physical clock synchronization- Keeping them synchronized is a fundamental challenge
- Leader election- A leader is necessary even for many distributed algorithms
- Mutual exclusion -access to the critical resource(s) has to be coordinated
- Deadlock detection and resolution- Deadlock detection should be coordinated to avoid duplicate work, and deadlock resolution should be coordinated to avoid unnecessary aborts of processes.
- Termination detection- cooperation among the processes
- Garbage collection - objects that are no longer in use and that are not pointed to by any other process.

RAMAIAH Institute of Technology

Design issues and challenges

5.Group communication, multicast, and ordered message delivery

- A group is a collection of processes that share a common context and collaborate on a common task within an application domain.
- Specific algorithms need to be designed to enable efficient group communication and group management wherein processes can join and leave groups dynamically, or even fail.
- When multiple processes send messages concurrently, different recipients may receive the messages in different orders, possibly violating the semantics of the distributed program.
- Hence, formal specifications of the semantics of ordered delivery need to be formulated, and then implemented.

RAMAIAH Institute of Technology

Design issues and challenges

6.Monitoring distributed events and predicates

- Predicates defined on program variables that are local to different processes are used for specifying conditions on the global system state,
- Application - debugging, sensing the environment, industrial process control.
- On-line algorithms for monitoring such predicates are hence important.
- An important paradigm for monitoring distributed events is that of event streaming, wherein streams of relevant events reported from different processes are examined collectively to detect predicates.

Design issues and challenges

7.Distributed program design and verification tools

Methodically designed and verifiably correct programs can greatly reduce the overhead of

- software design,
- debugging,
- engineering.

Designing mechanisms to achieve these design and verification goals is a challenge.

Design issues and challenges

8.Debugging distributed programs

- Debugging sequential programs is hard;
- Debugging distributed programs is that much harder because of the concurrency in actions and the ensuing uncertainty due to the large number of possible executions defined by the interleaved concurrent actions.
- Adequate debugging mechanisms and tools need to be designed to meet this challenge.

Design issues and challenges

9.Data replication, consistency models, and caching

- Fast access to data and other resources requires them to be replicated in the distributed system.
- Managing such replicas in the face of updates introduces the problems of ensuring consistency among the replicas and cached copies.
- Additionally, placement of the replicas in the systems is also a challenge because resources usually cannot be freely replicated.

RAMAIAH Institute of Technology

Design issues and challenges

10. World Wide Web design – caching, searching, scheduling

- Example : Web - distributed system with a direct interface to the end user, wherein the operations are predominantly read-intensive on most objects.
- Object search and navigation on the web are important functions in the operation of the web, and are very resource-intensive.
- Designing mechanisms to do this efficiently and accurately is a great challenge.

RAMAIAH Institute of Technology

Design issues and challenges

11. Distributed shared memory abstraction

- A shared memory abstraction simplifies the task of the programmer because he or she has to deal only with read and write operations, and no message communication primitives.
- However, under the covers in the middleware layer, the abstraction of a shared address space has to be implemented by using message-passing.
- Hence, in terms of overheads, the shared memory abstraction is not less expensive.

Design issues and challenges

11. Distributed shared memory abstraction

Wait-free algorithms

Wait-freedom, which can be informally defined as the ability of a process to complete its execution irrespective of the actions of other processes

Mutual exclusion

A first course in operating systems covers the basic algorithms (such as the Bakery algorithm and using semaphores) for mutual exclusion in a multiprocessing (uniprocessor or multiprocessor) shared memory setting.

Design issues and challenges

11. Distributed shared memory abstraction

Register constructions

Example such as biocomputing and quantum computing – that can alter the present foundations of computer “hardware” design, we need to revisit the assumptions of memory access of current systems

Consistency models

For multiple copies of a variable/object, varying degrees of consistency among the replicas can be allowed.

These represent a trade-off of coherence versus cost of implementation

Design issues and challenges

11. Distributed shared memory abstraction

Wait-free algorithms

Wait-freedom, which can be informally defined as the ability of a process to complete its execution irrespective of the actions of other processes

Mutual exclusion

A first course in operating systems covers the basic algorithms (such as the Bakery algorithm and using semaphores) for mutual exclusion in a multiprocessing (uniprocessor or multiprocessor) shared memory setting.

Design issues and challenges

12.Reliable and fault-tolerant distributed systems

Can be addressed using various strategies:

- Consensus algorithms
- Replication and replica management
- Voting and quorum systems
- Distributed databases and distributed commit
- Self-stabilizing systems
- Checkpointing and recovery algorithms
- Failure detectors

RAMAIAH Institute of Technology

Design issues and challenges

12.Reliable and fault-tolerant distributed systems

Consensus algorithms -Consensus algorithms allow correctly functioning processes to reach agreement among themselves in spite of the existence of some malicious (adversarial) processes whose identities are not known to the correctly functioning processes.

Replication and replica management -Replication is a classical method of providing fault-tolerance

Voting and quorum systems -Providing redundancy in the active (e.g., processes) or passive (e.g., hardware resources) components in the system and then performing voting based on some quorum criterion is a classical way of dealing with fault-tolerance.

RAMAIAH Institute of Technology

Design issues and challenges

12.Reliable and fault-tolerant distributed systems

Distributed databases and distributed commit- For distributed databases, the traditional properties of the transaction (A.C.I.D. – atomicity, consistency, isolation, durability) need to be preserved in the distributed setting

Self-stabilizing systems - A self-stabilizing algorithm is any algorithm that is guaranteed to eventually take the system to a good state even if a bad state were to arise due to some error.

RAMAIAH Institute of Technology

Design issues and challenges

12.Reliable and fault-tolerant distributed systems

Checkpointing and recovery algorithms- Checkpointing involves periodically recording the current state on secondary storage so that, in case of a failure, the entire computation is not lost but can be recovered

Failure detectors - Failure detectors represent a class of algorithms that probabilistically suspect another process as having failed and then converge on a determination of the up/down status of the suspected process.

Design issues and challenges

12. Load balancing

The goal of load balancing is to gain higher throughput, and reduce the user-perceived latency.

Load balancing may be necessary because of a variety of factors such as

- high network traffic
- high request rate causing the network connection to be a bottleneck
- high computational load

RAMAIAH Institute of Technology

Design issues and challenges

12. Load balancing

The following are some forms of load balancing:

- Data migration -The ability to move data (which may be replicated) around in the system, based on the access pattern of the users.
- Computation migration -The ability to relocate processes in order to perform a redistribution of the workload.
- Distributed scheduling -This achieves a better turnaround time for the users by using idle processing power in the system more efficiently.

Design issues and challenges

13.Real-time scheduling

Real-time scheduling is important for mission-critical applications, to accomplish the task execution on schedule.

The problem becomes more challenging in a distributed system where a global view of the system state is absent.

On-line or dynamic changes to the schedule are also harder to make without a global view of the state.

RAMAIAH Institute of Technology

Design issues and challenges

14 Performance

In large distributed systems, network latency (propagation and transmission times) and access to shared resources can lead to large delays which must be minimized.

- Metrics - Appropriate metrics must be defined or identified for measuring the performance of theoretical distributed algorithms, as well as for implementations of such algorithms.
- Measurement methods/tools - As a real distributed system is a complex entity and has to deal with all the difficulties that arise in measuring performance over a WAN/the Internet, appropriate methodologies and tools must be developed for measuring the performance metrics.

Design issues and challenges

Applications of distributed computing and newer challenges

1. **Mobile systems**
2. **Sensor networks**
3. **Ubiquitous or pervasive computing**
4. **Publish-subscribe, content distribution, and multimedia**
5. **Distributed agents**
6. **Distributed data mining**
7. **Grid computing**
8. **Security in distributed systems**

RAMAIAH Institute of Technology

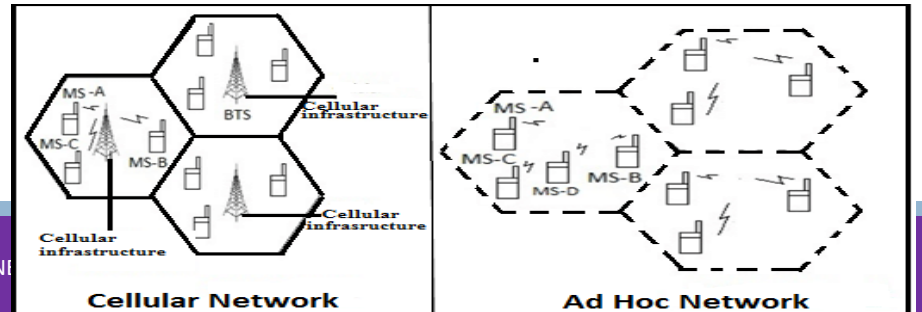
Design issues and challenges

Applications of distributed computing and newer challenges

1.Mobile systems

There are two popular architectures for a mobile network

- Base-station (cellular approach) a cell which is the geographical region within range of a static but powerful base transmission station is associated with that base station.
- Ad-hoc network approach where there is no base station (which essentially acted as a centralized node for its cell).



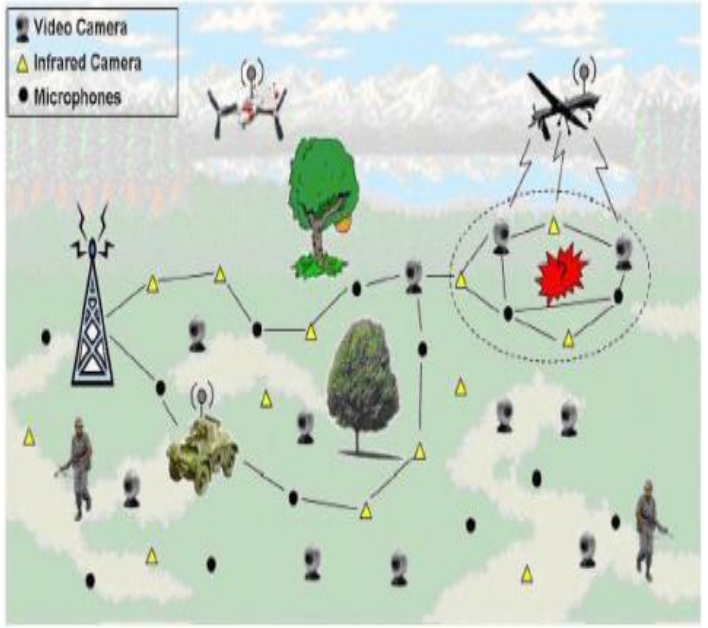
RAMAIAH Institute of Technology

Design issues and challenges

Applications of distributed computing and newer challenges

2.Sensor networks

- A sensor is a processor with an electro-mechanical interface that is capable of sensing physical parameters, such as temperature, velocity, pressure, humidity, and chemicals.
- Sensors may be mobile or static; sensors may communicate wirelessly, although they may also communicate across a wire when they are statically installed.
- Sensors may have to self-configure to form an ad-hoc network, which introduces a whole new set of challenges, such as position estimation and time estimation.



RAMAIAH Institute of Technology

Design issues and challenges

Applications of distributed computing and newer challenges

3.Ubiquitous or pervasive computing

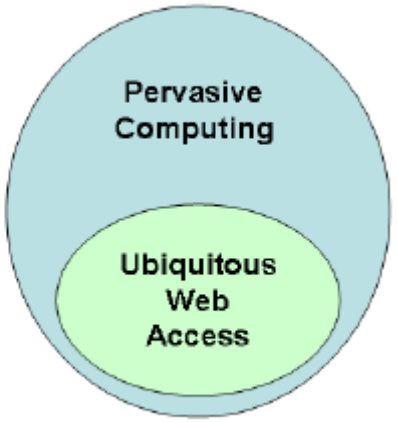
Ubiquitous systems are essentially distributed systems

Ubiquitous systems represent a class of computing where the processors embedded in and seamlessly pervading through the environment perform application functions

Example:

- The intelligent home
- The smart workplace

- Pervasive computing
  - Access to information and services
  - Focus on everyday's activity
  - Any user
  - Any device
  - Anytime
- Ubiquitous Web access
  - Access to **Web-based** information and services
  - Subset of pervasive computing
  - Any user
  - Any device
  - Anytime



Design issues and challenges

Applications of distributed computing and newer challenges

4.Publish-subscribe, content distribution, and multimedia

Example - Stock prices

In a dynamic environment where the information constantly fluctuates (there needs to be:

(i) an efficient mechanism for distributing this information (publish),

(ii) (ii) an efficient mechanism to allow end users to indicate interest in receiving specific kinds of information (subscribe), and

(iii) (iii) an efficient mechanism for aggregating large volumes of published information and filtering it as per the user's subscription filter.

UNIT -1

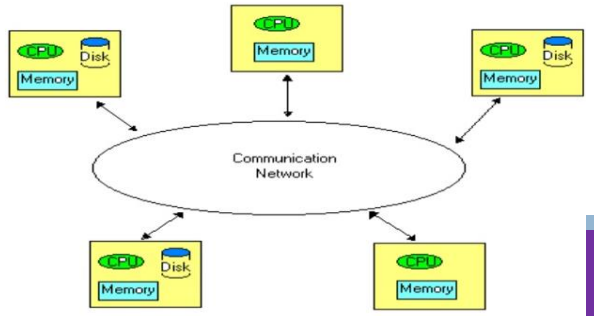
A model of distributed computations:(Chapter 2 – 2.1,2.2,2.3,2.4,2.5,2.6)

- A distributed program
- A model of distributed executions
- Models of communication networks
- Global state of a distributed system
- Cuts of a distributed computation
- Past and future cones of an event

RAMAIAH Institute of Technology

Introduction

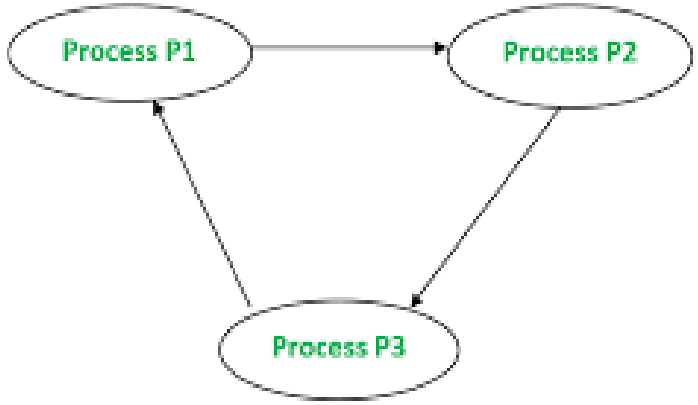
- A distributed system consists of a set of processors that are connected by a communication network.
- The communication network provides the facility of information exchange among processors.
- The processors do not share a common global memory and communicate solely by passing messages over the communication network.
- There is no physical global clock in the system to which processes have instantaneous access.



Introduction

The communication medium may deliver messages

- out of order,
- messages may be lost,
- garbled
- duplicated due to timeout and retransmission,
- processors may fail
- communication links may go down.



The system can be modeled as a directed graph in which vertices represent the processes and edges represent unidirectional communication channels.

A distributed application runs as a collection of processes on a distributed system.

A distributed program

- A distributed program is composed of a set of n asynchronous processes p1, p2,..pi,. pn that communicate by message passing over the communication network.
- Assume that each process is running on a different processor.
- Let C_{ij} denote the channel from process p_i to process p_j
- Let m_{ij} denote a message sent by pi to pj .
- Process execution and message transfer are asynchronous – a process may execute an action spontaneously and a process sending a message does not wait for the delivery of the message to be complete.

A distributed program

The global state of a distributed computation is composed of the

- State of the processes
- State of the communication channels .

- The state of a process is characterized by the state of its local memory and depends upon the context.
- The state of a channel is characterized by the set of messages in transit in the channel.

UNIT -1

A model of distributed computations:(Chapter 2 – 2.1,2.2,2.3,2.4,2.5,2.6)

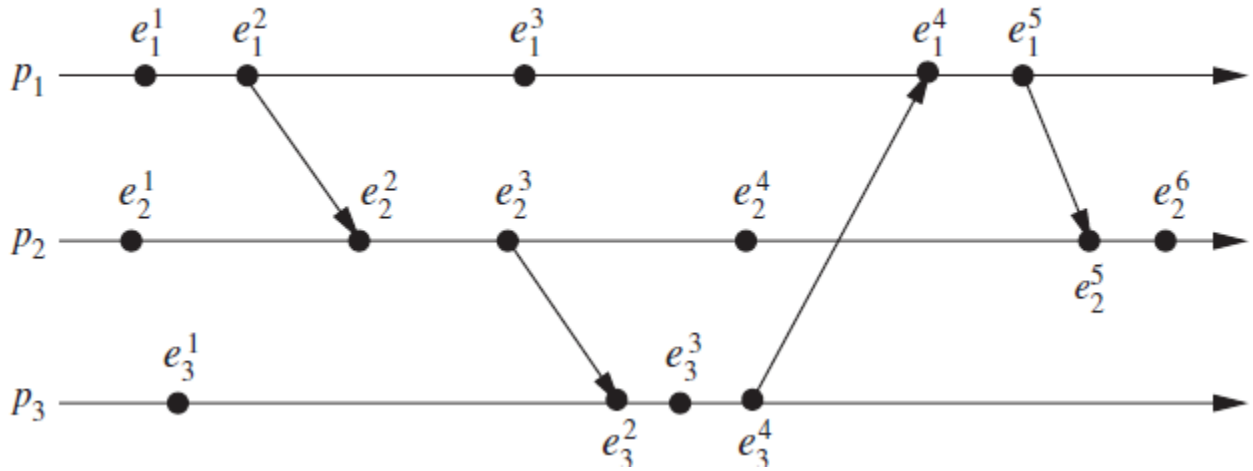
- A distributed program
- A model of distributed executions
- Models of communication networks
- Global state of a distributed system
- Cuts of a distributed computation
- Past and future cones of an event

A model of distributed executions

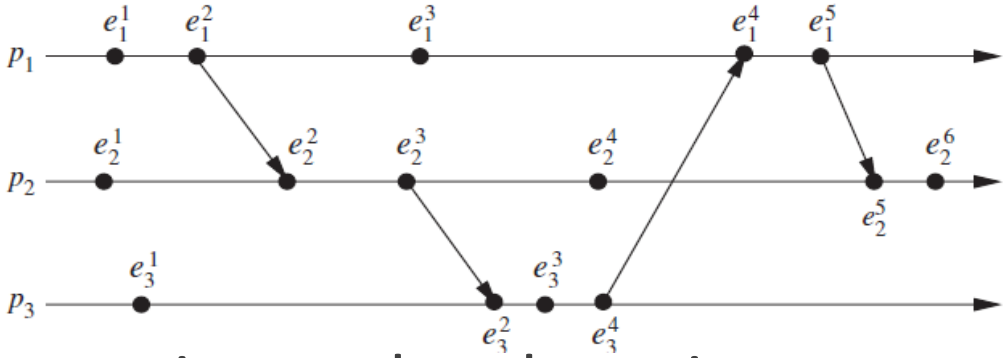
The execution of a process consists of a sequential execution of its actions.

The actions are atomic and the actions of a process are modeled as three types of events:

- internal events,
- message send events,
- message receive events.



A model of distributed



- Let e^x_i denote the x^{th} event at process p_i
- For a message m, let send(m) and rec(m)denote its send and receive events, respectively.
- The occurrence of events changes the states of respective processes and channels, thus causing transitions in the global system state.
- An internal event changes the state of the process at which it occurs.
- A send event (or a receive event) changes the state of the process that sends (or receives) the message and the state of the channel on which the message is sent (or received).

A model of distributed executions

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

The execution of process p_i produces a sequence of events

$$\mathcal{H}_i := e_i^1, e_i^2, \ldots, e_i^x, e_i^{x+1}, \ldots$$

$$\mathcal{H}_i = (h_i, \rightarrow_i)$$

where h_i is the set of events produced by p_i

binary relation $\rightarrow_i$ defines a linear order on these events

Relation $\rightarrow_i$ expresses causal dependencies among the events of p_i.

A model of distributed executions

A relation $\rightarrow_{msg}$ that captures the causal dependency due to message exchange, is defined as follows.

For every message m that is exchanged between two processes, we have

$$send(m) \rightarrow_{msg} rec(m).$$

A model of distributed executions

The evolution of a distributed execution is depicted by a space–time diagram involving three processes.

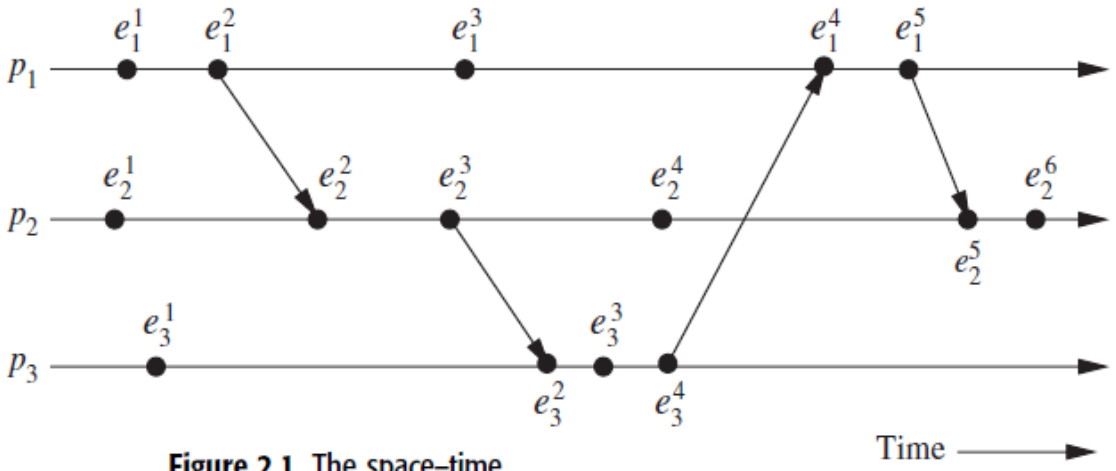


Figure 2.1 The space–time diagram of a distributed execution.

- A horizontal line represents the progress of the process
- a dot indicates an event;
- a slant arrow indicates a message transfer.

A model of distributed executions

Causal precedence relation

The execution of a distributed application results in a set of distributed events produced by the processes.

Let $H = \cup_i h_i$ denote the set of events executed in a distributed computation.

A model of distributed executions

Causal precedence relation

Define a binary relation on the set H, denoted as →, that expresses causal dependencies between events in the distributed execution.

$$\forall e_i^x,\ \forall e_j^y \in H,\quad e_i^x \rightarrow e_j^y \quad \Leftrightarrow \quad \begin{cases} e_i^x \rightarrow_i e_j^y \text{ i.e., } (i = j) \wedge (x < y) \\ \text{or} \\ e_i^x \rightarrow_{msg} e_j^y \\ \text{or} \\ \exists e_k^z \in H : e_i^x \rightarrow e_k^z \ \wedge \ e_k^z \rightarrow e_j^y \end{cases}$$

RAMAIAH Institute of Technology

A model of distributed executions

The causal precedence relation induces an irreflexive partial order on the events of a distributed computation that is denoted as

H=(H, →)

$$\mathcal{H}_i := e_i^1,\ e_i^2,\ \ldots,\ e_i^x, e_i^{x+1}, \ldots$$

$$\mathcal{H}_i = (h_i, \rightarrow_i)$$

$$\forall e_i^x,\ \forall e_j^y \in H,\quad e_i^x \rightarrow e_j^y \quad \Leftrightarrow \quad \begin{cases} e_i^x \rightarrow_i e_j^y \text{ i.e., } (i = j) \wedge (x < y) \\ \text{or} \\ e_i^x \rightarrow_{msg} e_j^y \\ \text{or} \\ \exists e_k^z \in H : e_i^x \rightarrow e_k^z \ \wedge \ e_k^z \rightarrow e_j^y \end{cases}$$

A model of distributed

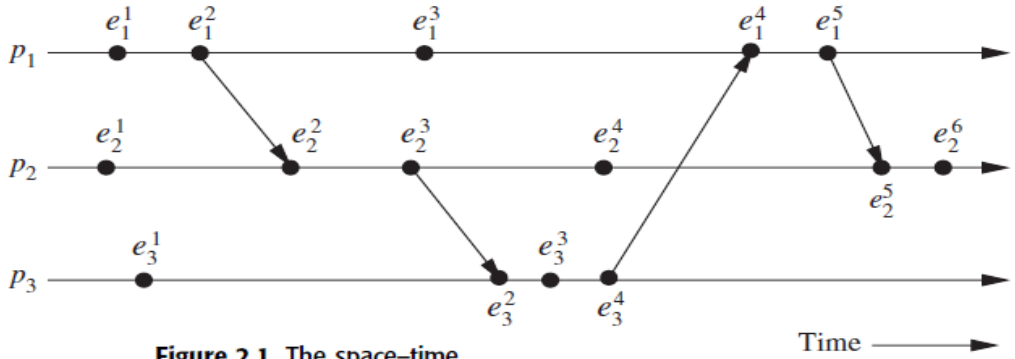


Figure 2.1 The space–time diagram of a distributed execution.

For any two events e_i and e_j,

if $e_i \rightarrow e_j$, then event ej is directly or transitively dependent on event ei; graphically, it means that there exists a path consisting of message arrows and process-line segments (along increasing time) in the space–time diagram that starts at ei and ends at ej .

Example: $e^1_1 \rightarrow e^3_3$ and $e^3_3 \rightarrow e^6_2$

Note that relation → denotes flow of information in a distributed computation and ei→ej dictates that all the information available at ei is potentially accessible at ej .

Event e^6_2 has the knowledge of all other events

A model of distributed executions

For any two events e_i and e_j,

Note that relation || is not transitive

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 said to be concurrent and the relation is denoted as $e_i \parallel e_j$.

Example: $e^3_1 \parallel e^3_3$ and $e^4_2 \parallel e^1_3$

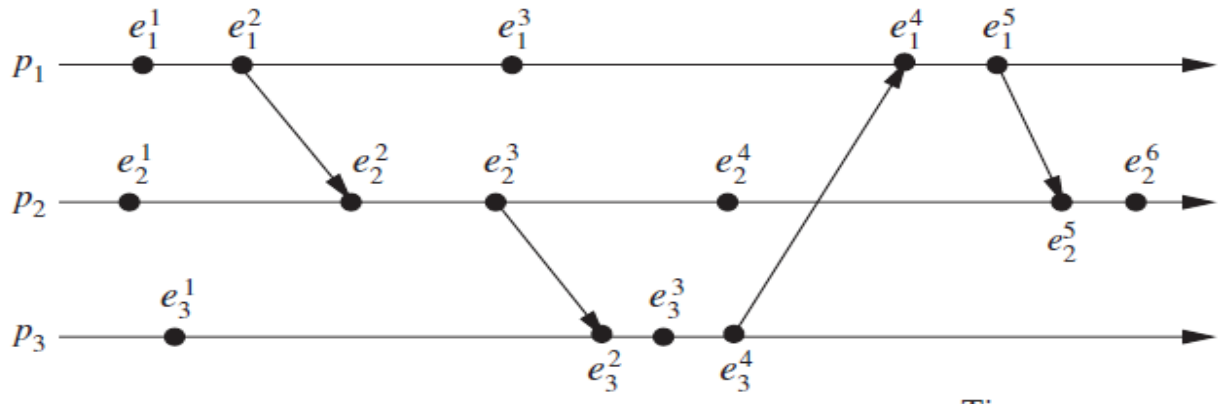


Figure 2.1 The space–time diagram of a distributed execution.

UNIT -1

A model of distributed computations:(Chapter 2 – 2.1,2.2,2.3,2.4,2.5,2.6)

- A distributed program
- A model of distributed executions
- Models of communication networks
- Global state of a distributed system
- Cuts of a distributed computation
- Past and future cones of an event

Models of communication networks

There are several models of the service provided by communication networks, namely,

1. FIFO (first-in, first-out)
2. non-FIFO
3. causal ordering

FIFO model, each channel acts as a first-in first-out message queue and thus, message ordering is preserved by a channel.

non-FIFO model, a channel acts like a set in which the sender process adds messages and the receiver process removes messages from it in a random order.

Models of communication networks

Causal ordering model is based on Lamport's "happens before" relation.

A system that supports the causal ordering model satisfies the following property:

CO: For any two messages m_{ij} and m_{kj}, if $send(m_{ij}) \longrightarrow send(m_{kj})$,
then $rec(m_{ij}) \longrightarrow rec(m_{kj})$.

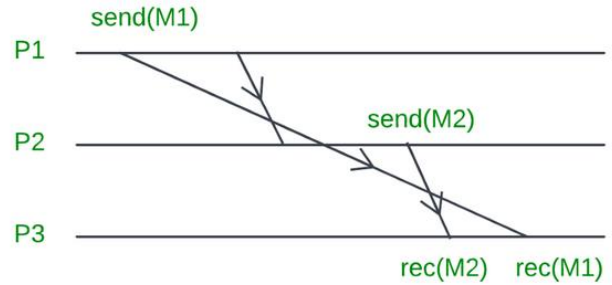
Causal ordering model is useful in developing distributed algorithms

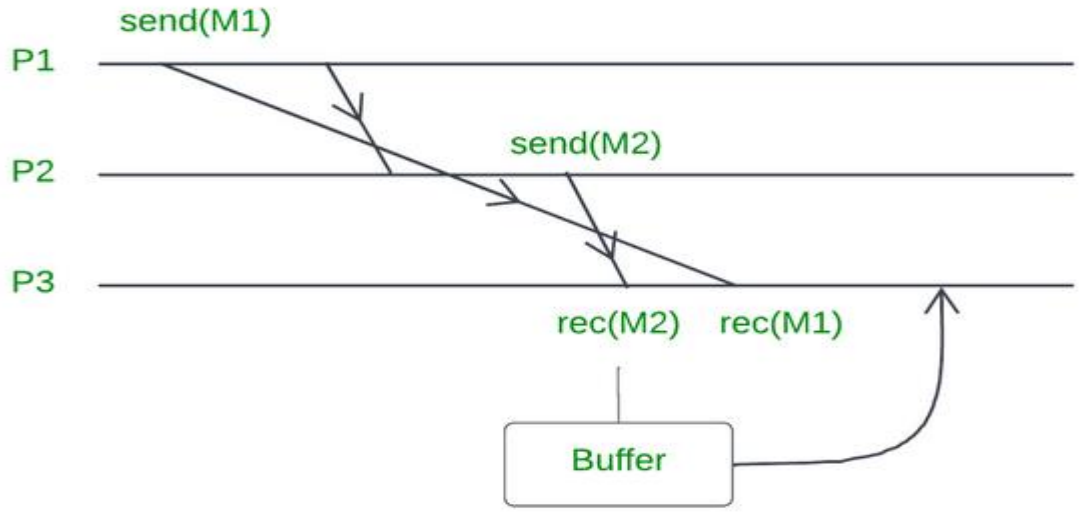
Models of communication networks

The causal ordering of messages describes the causal relationship between a message send event and a message receive event.

CO: For any two messages m_{ij} and m_{kj}, if $send(m_{ij}) \longrightarrow send(m_{kj})$,
then $rec(m_{ij}) \longrightarrow rec(m_{kj})$.

For example,
if send(M1) -> send(M2) then every recipient of both the messages M1 and M2 must receive the message M1 before receiving the message M2.





Models of communication networks

- In a distributed computation, two events are logically concurrent if and only if they do not causally affect each other.
- Physical concurrency has a connotation that the events occur at the same instant in physical time.
- Two or more events may be logically concurrent even though they do not occur at the same instant in physical time.

UNIT -1

A model of distributed computations:(Chapter 2 – 2.1,2.2,2.3,2.4,2.5,2.6)

- A distributed program
- A model of distributed executions
- Models of communication networks
- Global state of a distributed system
- Cuts of a distributed computation
- Past and future cones of an event

Global state of a distributed system

Garbage Collector

- Frees up memory which is no longer in use
- Checks if a reference to memory still exists

What about in a distributed system?

- A distributed system consists of multiple processes
- Each process is located on a different computer
- No sharing of processor or memory

Each process can only determine its own state Problem

- How do we determine when to garbage collect in a distributed system?
- How do we check whether a reference to memory still exists?

Global state of a distributed system

- A distributed system consists of multiple processes
- Each process is located on a different computer
- Each process consists of events
- An event is either sending a message, receiving a message, or changing the value of some variable
- Each process has a communication channel in and out

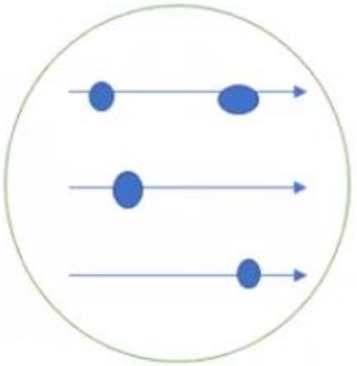
A snapshot of the entire system must be taken to test whether a certain property of the system is true. This snapshot is called a **Global State**

Global state of a distributed system

What is Global Snapshot?

- Real time Scenario : Representatives from different country captured in one photograph.
- Imagine : Representatives sitting in their own country and sending messages → Distributed snapshot
- Challenge is to calculate global snapshot.
- Global snapshot = global state
- Solution : Synchronization (however it leads to errors.

Consider bank informing that due to clock skew the bank balance is now nil.

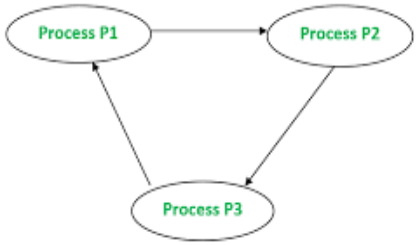


RAMAIAH Institute of Technology

Global state of a distributed system

"A collection of the local states of its components, namely, the processes and the communication channels."

- The state of a process is defined by the contents of processor registers,stacks, local memory, etc. and depends on the local context of the distributed application.
- The state of channel is given by the set of messages in transit in the channel.
- The occurrence of events changes the states of respective processes and channels.
- An internal event changes the state of the process at which it occurs.



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

Global state of a distributed system

Notations

- LS_i^x denotes the state of process p_i after the occurrence of event e_i^x and before the event e_i^{x+1}.
- LS_i^0 denotes the initial state of process p_i.
- LS_i^x is a result of the execution of all the events executed by process p_i till e_i^x.
- Let $send(m) \leq LS_i^x$ denote the fact that $\exists y{:}1 \leq y \leq x :: e_i^y = send(m)$.
- Let $rec(m) \not\leq LS_i^x$ denote the fact that $\forall y{:}1 \leq y \leq x :: e_i^y \neq rec(m)$.

Global state of a distributed system

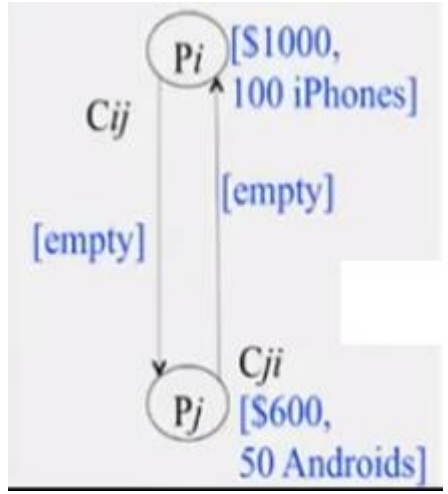
A Channel State

- The state of a channel depends upon the states of the processes it connects.
- Let $SC_{ij}^{x,y}$ denote the state of a channel C_{ij}.

The state of a channel is defined as follows:

$$SC_{ij}^{x,y} = \{m_{ij} | \ send(m_{ij}) \leq e_i^x \bigwedge rec(m_{ij}) \not\leq e_j^y\}$$

Thus, channel state $SC_{ij}^{x,y}$ denotes all messages that p_i sent upto event e_i^x and which process p_j had not received until event e_j^y.



Global state of a distributed system

Global State

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

$$GS = \{\bigcup_i LS_i^{x_i}, \bigcup_{j,k} SC_{jk}^{y_j,z_k}\}$$

- For a global state to be meaningful, the states of all the components of the distributed system must be recorded at the same instant.
- This will be possible if the local clocks at processes were perfectly synchronized or if there were a global system clock that can be instantaneously read by the processes. (However, both are impossible.)

Global state of a distributed system

A Consistent Global State

- Basic idea is that a state should not violate causality – an effect should not be present without its cause.
- A message cannot be received if it was not sent. Such states are called consistent global states
- Inconsistent global states are not meaningful in the sense that a distributed system can never be in an inconsistent state.

RAMAIAH
Institute of Technology

Global state of a distributed system

A Consistent Global State

- A global state $GS = \{\bigcup_i LS_i^{x_i}, \bigcup_{j,k} SC_{jk}^{y_j,z_k}\}$ is a *consistent global state* iff

$$\forall m_{ij}: \; send(m_{ij}) \notin LS_i^{x_i} \;\Leftrightarrow\; m_{ij} \notin SC_{ij}^{x_i,y_j} \wedge rec(m_{ij}) \notin LS_j^{y_j}$$

- That is, channel state $SC_{ij}^{y_i,z_k}$ and process state $LS_j^{z_k}$ must not include any message that process p_i sent after executing event $e_i^{x_i}$.

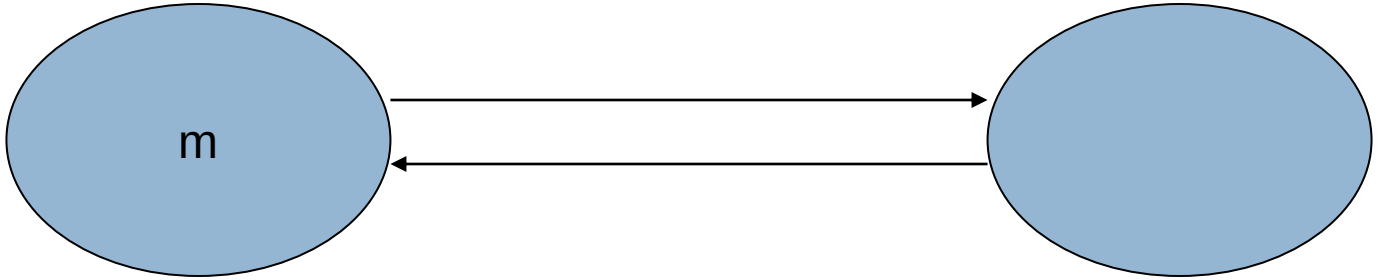
- A global state *GS* is a ***consistent global state*** iff it satisfies the following two conditions:

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

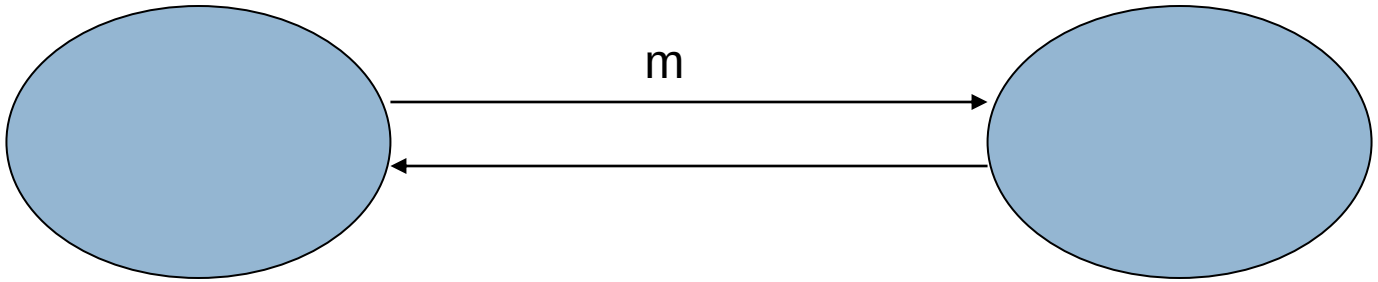
($\oplus$ is Ex-OR operator)

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

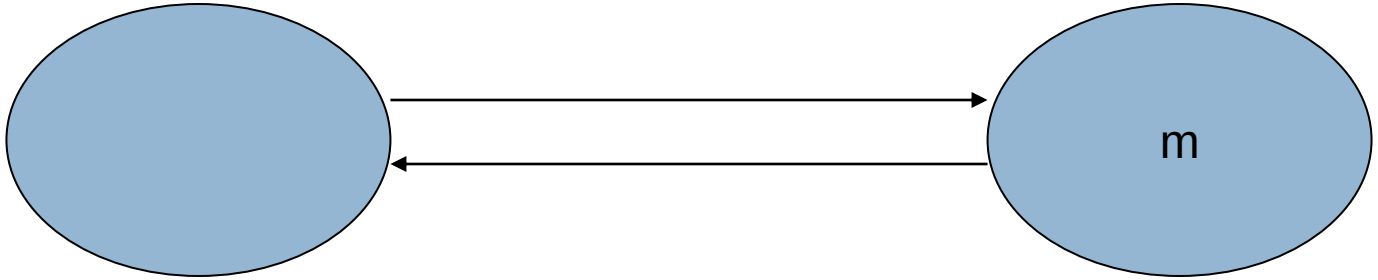
Example: Initial State



Example



Example



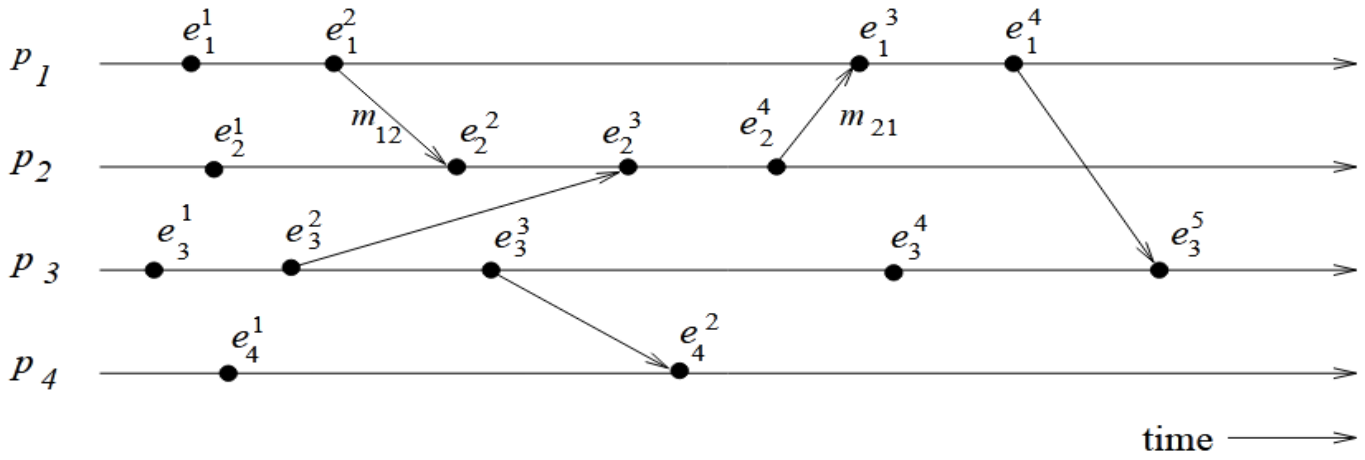
Global state of a distributed system

Let e_i^x denote the xth event at process p_i.

- LS_i^x is a result of the execution of all the events executed by process p_i till e_i^x.

An Example
Consider the distributed execution of Figure 2.2.

Figure 2.2: The space-time diagram of a distributed execution.

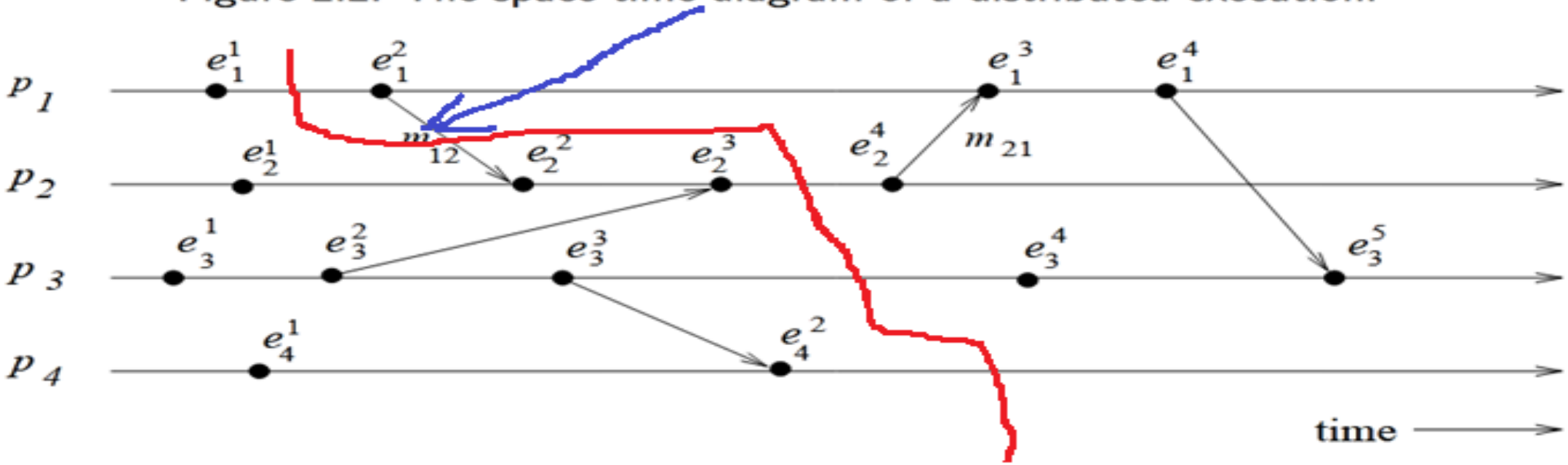


LS_1^2

An Example

Consider the distributed execution of Figure 2.2.

Figure 2.2: The space-time diagram of a distributed execution.



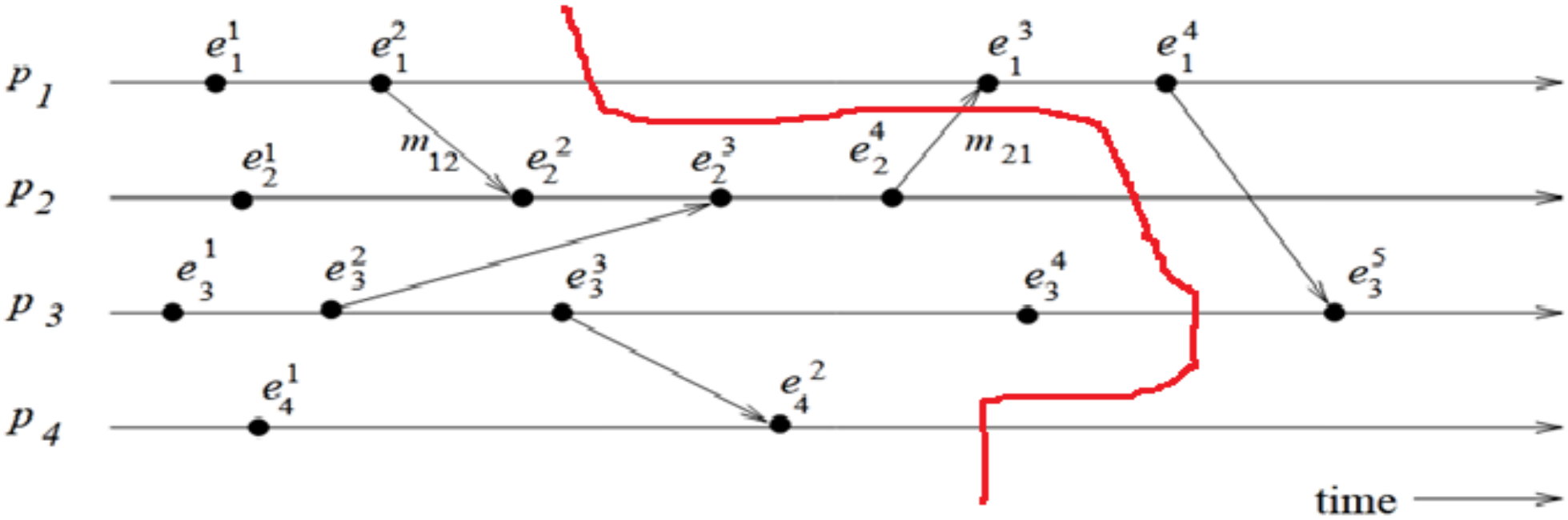
$GS_1 = \{LS_1^1, LS_2^3, LS_3^3, LS_4^2\}$ is inconsistent

because the state of p_2 has recorded the receipt of message m_{12}, however, the state of p_1 has not recorded its send.

An Example

Consider the distributed execution of Figure 2.2.

Figure 2.2: The space-time diagram of a distributed execution.



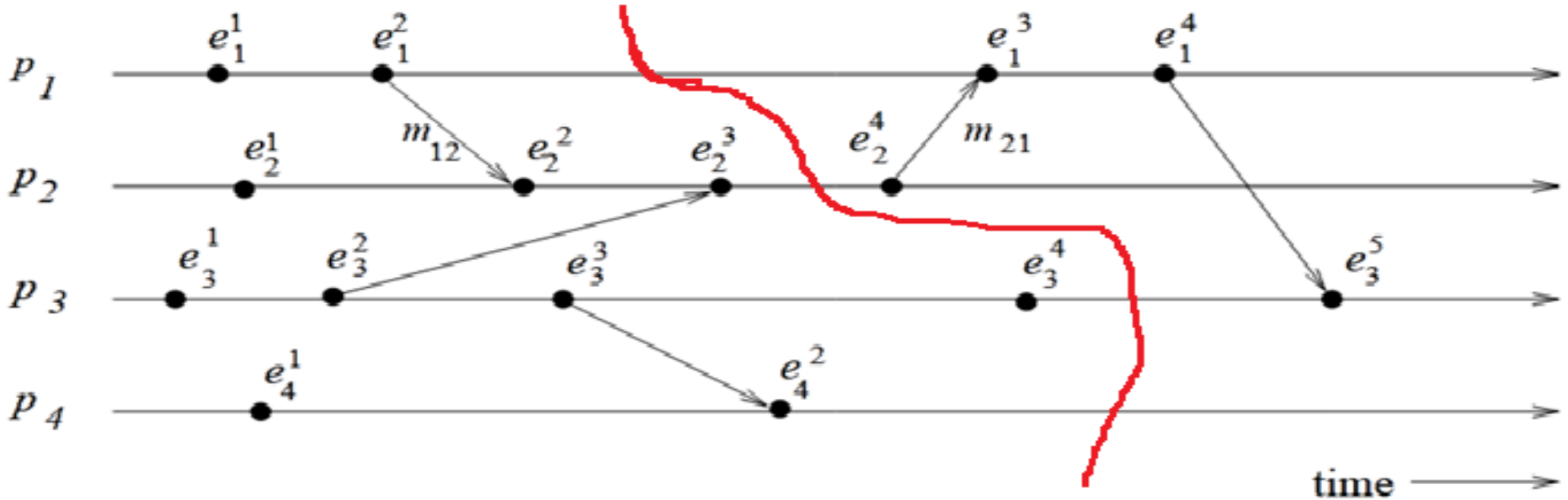
- A global state GS_2 consisting of local states $\{LS_1^2, LS_2^4, LS_3^4, LS_4^2\}$ is consistent; all the channels are empty except C_{21} that contains message m_{21}.

Global state of a distributed system

An Example

Consider the distributed execution of Figure 2.2.

Figure 2.2: The space-time diagram of a distributed execution.



$GS_3 = LS_1^2\ LS_2^3\ LS_3^4\ LS_4^2$ **Strongly consistent**

UNIT -1

A model of distributed computations:(Chapter 2 – 2.1,2.2,2.3,2.4,2.5,2.6)

- A distributed program
- A model of distributed executions
- Models of communication networks
- Global state of a distributed system
- Cuts of a distributed computation
- Past and future cones of an event

Cuts of a distributed computation

- "In the space-time diagram of a distributed computation, a cut is a zigzag line joining one arbitrary point on each process line."
- A cut slices the space-time diagram, and thus the set of events in the distributed computation, into a PAST and a FUTURE.

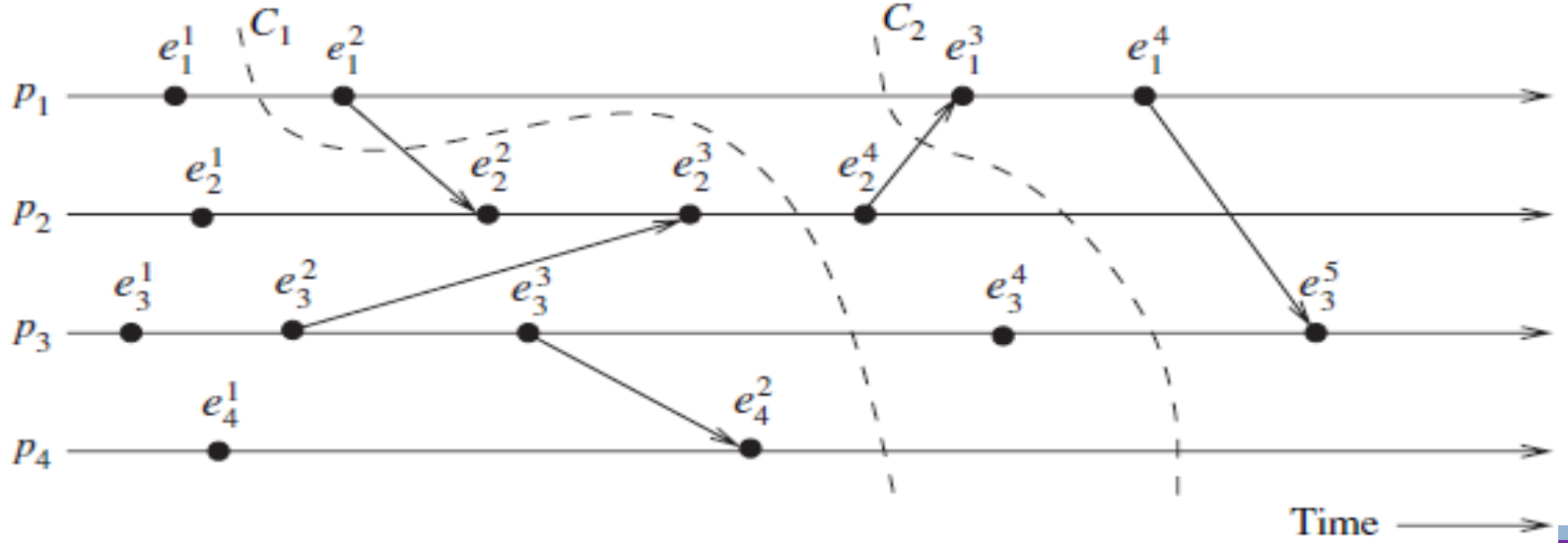


Figure 2.3 Illustration of cuts in a distributed execution.

Cuts of a distributed computation

- "The PAST contains all the events to the left of the cut and the FUTURE contains all the events to the right of the cut.
- For a cut C, let PAST(C) and FUTURE(C) denote the set of events in the PAST and FUTURE of C, respectively.
- Every cut corresponds to a global state and every global state can be graphically represented as a cut in the computation's space–time diagram

Cuts of a distributed computation

Types of cut

1. Consistent cut
2. Inconsistent cut

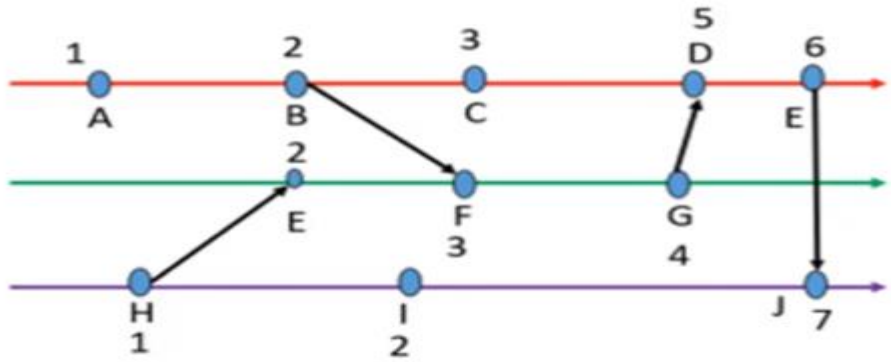
RAMAIAH Institute of Technology

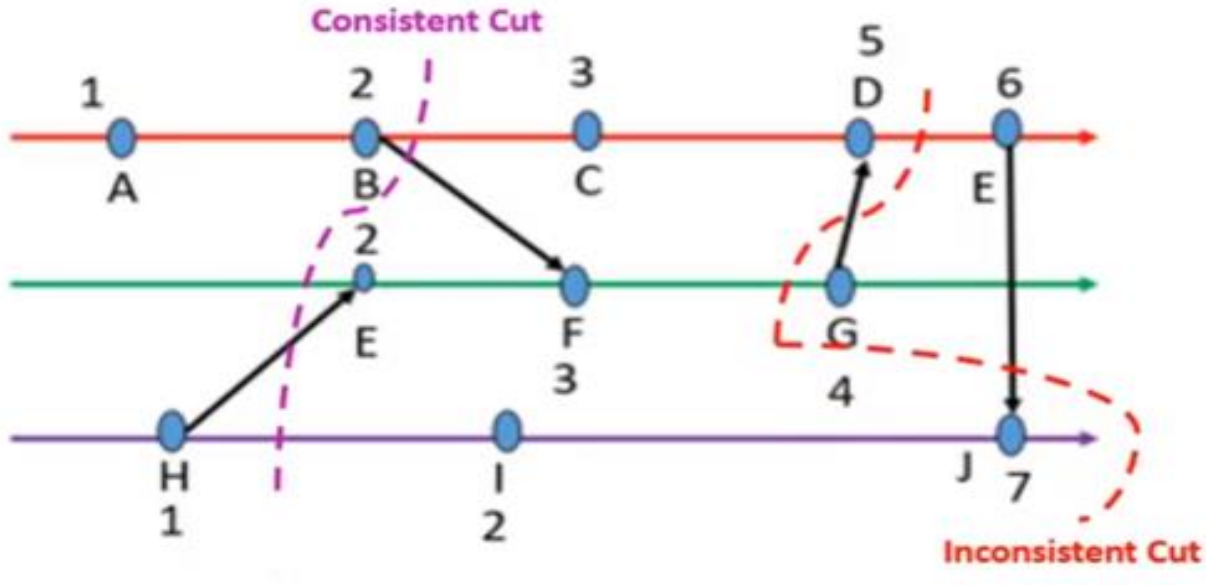
Cuts of a distributed computation

Consistent cut

A consistent global state corresponds to a cut in which every message received in the PAST of the cut was sent in the PAST of that cut. Such a cut is known as a consistent cut.

All messages that cross the cut from the PAST to the FUTURE are in transit in the corresponding consistent global state.



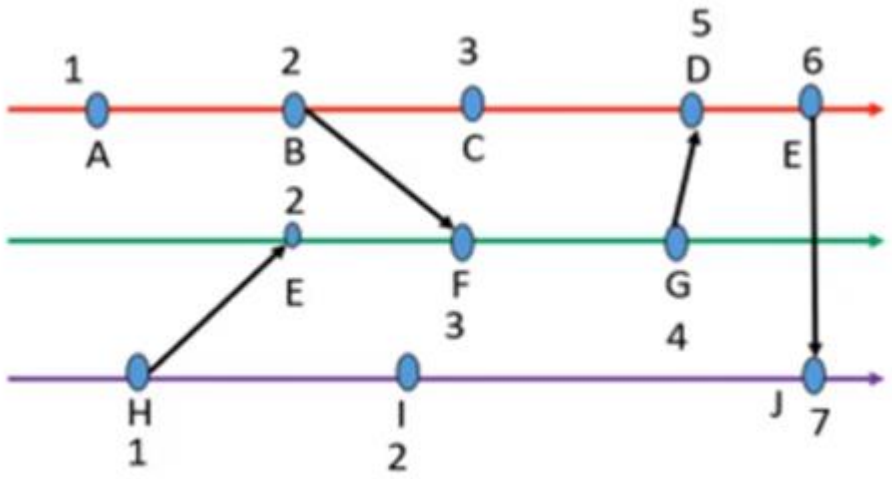


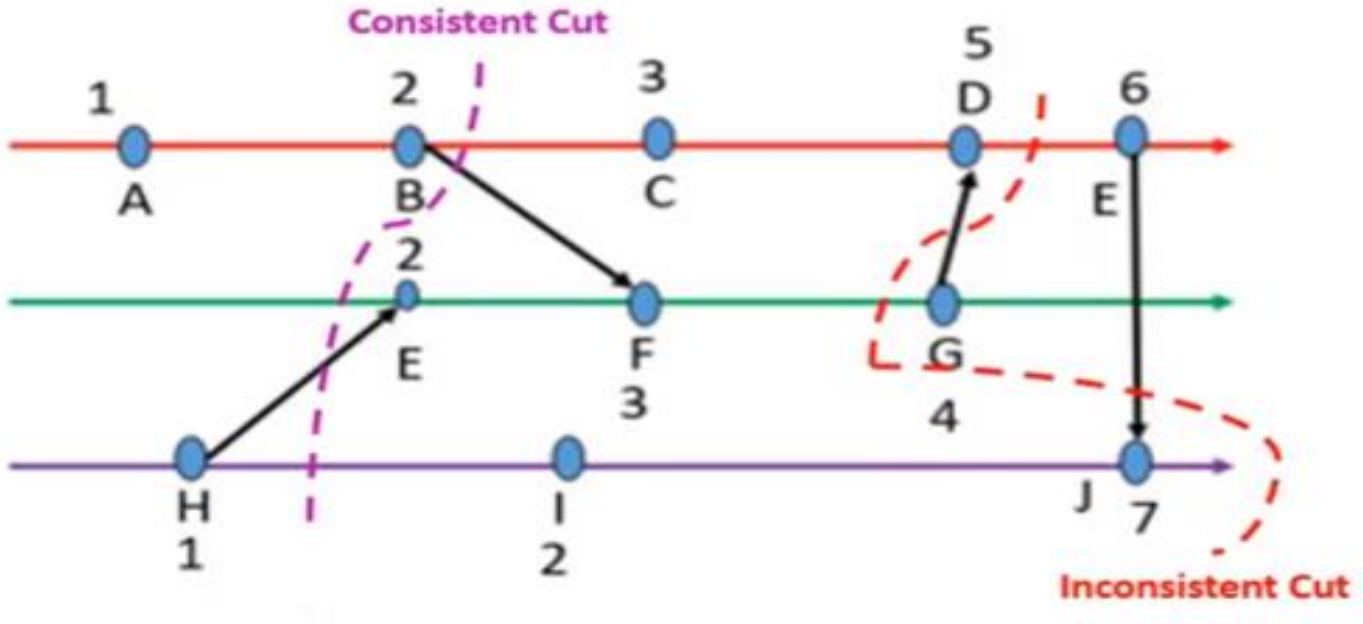
RAMAIAH
Institute of Technology

Cuts of a distributed computation

Inconsistent cut

A cut is inconsistent if a message crosses the cut from the FUTURE to the PAST.





Cuts of a distributed computation

- In a consistent cut, every message received in the PAST of the cut was sent in the PAST of that cut.
- Example : cut C2 is a consistent cut

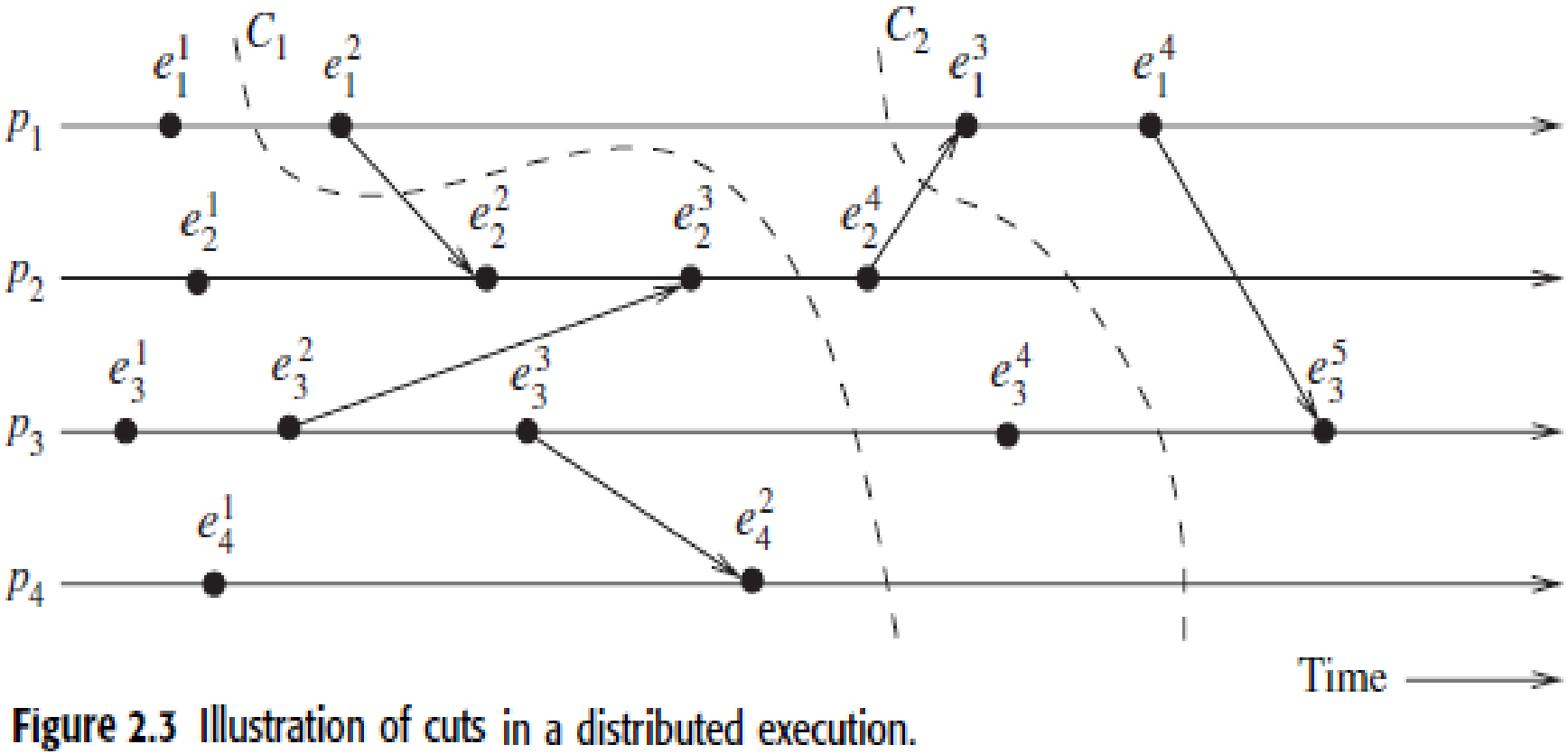


Figure 2.3 Illustration of cuts in a distributed execution.

Cuts of a distributed computation

- A cut is inconsistent if a message crosses the cut from the FUTURE to the PAST.
- Example :cut C1 is an inconsistent cut.)

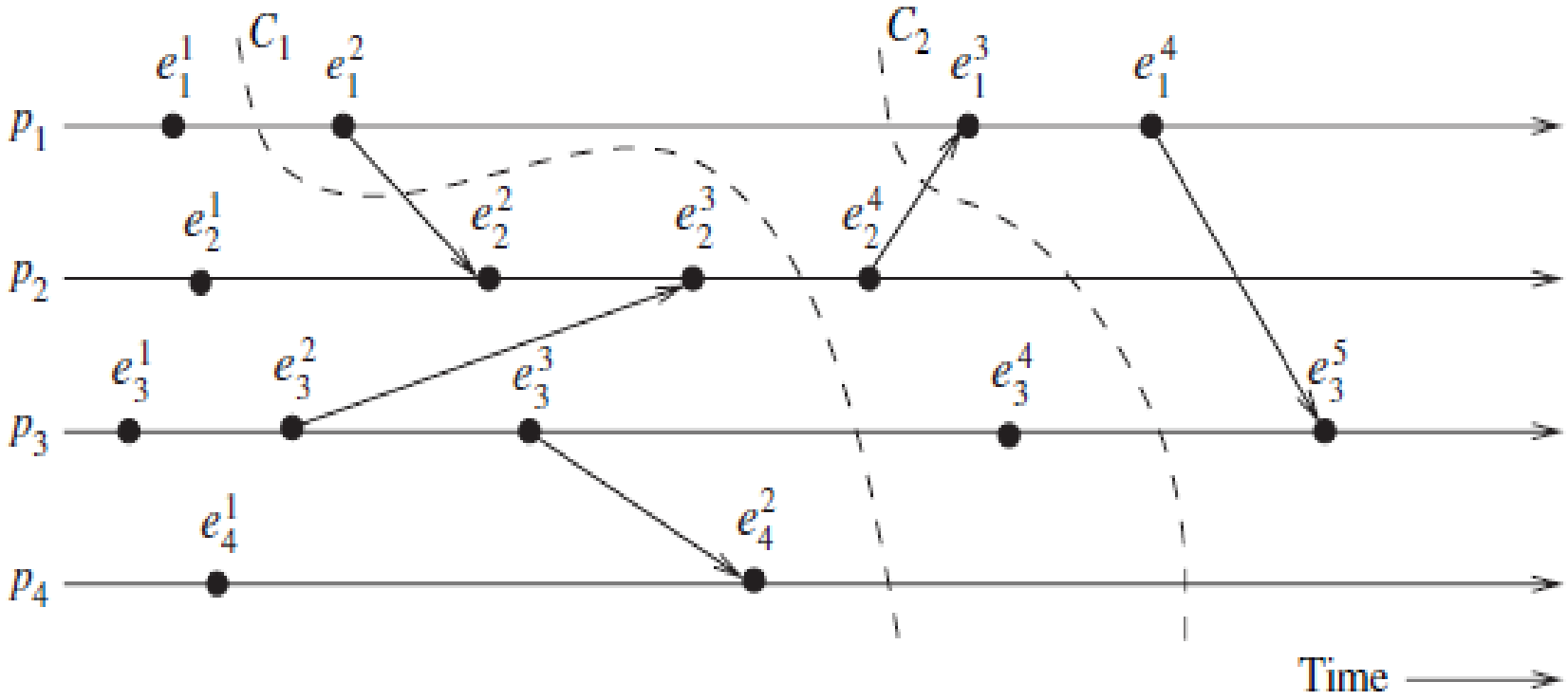


Figure 2.3 Illustration of cuts in a distributed execution.

UNIT -1

A model of distributed computations:(Chapter 2 – 2.1,2.2,2.3,2.4,2.5,2.6)

- A distributed program
- A model of distributed executions
- Models of communication networks
- Global state of a distributed system
- Cuts of a distributed computation
- Past and future cones of an event

Past and future cones of an event

Past Cone of an Event

- An event e_j could have been affected only by all events e_i such that $e_i \rightarrow e_j$.
- In this situtaion, all the information available at e_i could be made accessible at e_j.
- All such events e_i belong to the past of e_j.

Let $Past(e_j)$ denote all events in the past of e_j in a computation $(H, \rightarrow)$. Then,

$$Past(e_j) = \{e_i | \forall e_i \in H,\ e_i \rightarrow e_j\ \}.$$

RAMAIAH Institute of Technology

Past and future cones of an event

- Let $Past_i(e_j)$ be the set of all those events of $Past(e_j)$ that are on process p_i.
- $Past_i(e_j)$ is a totally ordered set, ordered by the relation $\rightarrow_i$, whose maximal element is denoted by $max(Past_i(e_j))$.
- $max(Past_i(e_j))$ is the latest event at process p_i that affected event e_j
- Let $Max\_Past(e_j) = \bigcup_{(\forall i)}\{max(Past_i(e_j))\}$.
- $Max\_Past(e_j)$ consists of the latest event at every process that affected event e_j and is referred to as the *surface of the past cone* of e_j.
- $Past(e_j)$ represents all events on the past light cone that affect e_j.

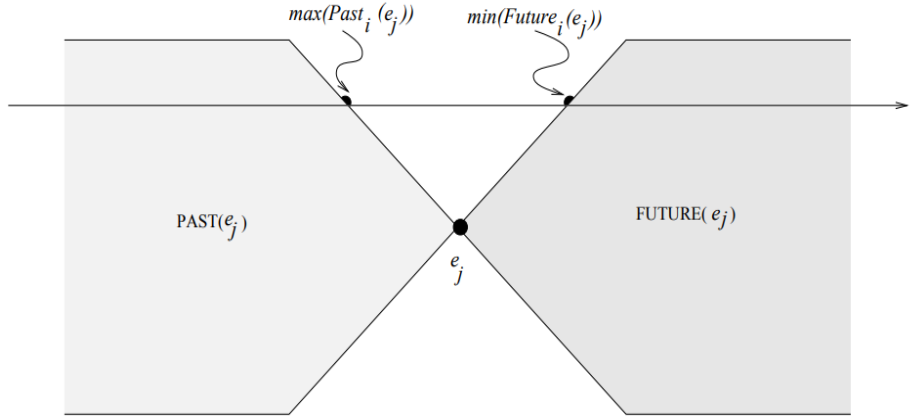


Figure 2.4 shows the past of an event e_j.

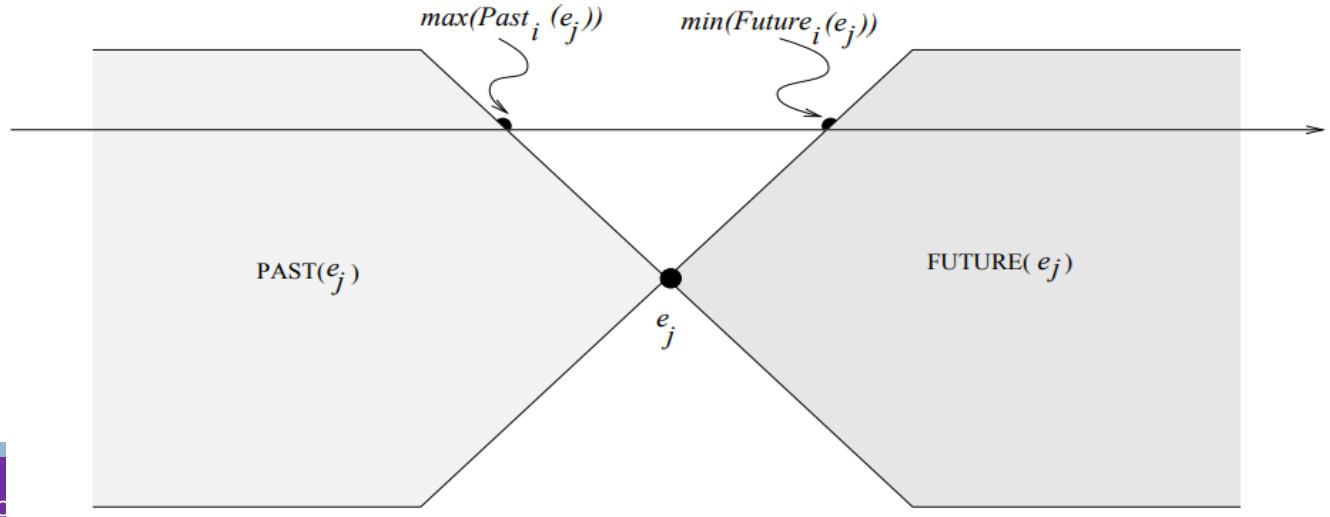
RAMAIAH Institute of Technology

Past and future cones of an event

Future Cone of an Event

- The future of an event e_j, denoted by $Future(e_j)$, contains all events e_i that are causally affected by e_j
- In a computation $(H, \rightarrow)$, $Future(e_j)$ is defined as:

$$Future(e_j) = \{e_i | \forall e_i \in H,\ e_j \rightarrow e_i\}.$$



Past and future cones of an event

- Define $Future_i(e_j)$ as the set of those events of $Future(e_j)$ that are on process p_i.
- define $min(Future_i(e_j))$ as the first event on process p_i that is affected by e_j.
- Define $Min\_Future(e_j)$ as $\bigcup_{(\forall i)}\{min(Future_i(e_j))\}$, which consists of the first event at every process that is causally affected by event e_j.
- $Min\_Future(e_j)$ is referred to as the *surface of the future cone* of e_j.
- All events at a process p_i that occurred after $max(Past_i(e_j))$ but before $min(Future_i(e_j))$ are concurrent with e_j.
- Therefore, all and only those events of computation H that belong to the set "$H - Past(e_j) - Future(e_j)$" are concurrent with event e_j.

UNIT -1

Logical time: (Chapter 3: 3.1,3.2,3.3,3.4,3.5,3.6)

- Introduction
- A framework for a system of logical clocks
- Scalar time
- Vector time
- efficient implementations of vector clocks
- Jard–Jourdan's adaptive technique

RAMAIAH Institute of Technology

Introduction

- Physical Clock is an electronic device that counts oscillations in a crystal at a particular frequency. Can calculate the timestamp of an event
- Logical clock refer to the implementation of a protocol on all machines, so that the machine are able to maintain consistent ordering of events with virtual timestamp(ordering of event is important)

Logical vs. Physical Clocks

- Logical clock keeps track of event ordering
  - Among related (causal) events
  - Do not care the real time where events occurred
- Physical clock keeps time of day
  - Consistent across systems

Introduction

Example

If we go outside then we have made a full plan that at which place we have to go first, second and so on. We don't go to second place at first and then the first place.

- 10 PCs are present in a distributed system and every PC is doing it's own work but then how we make them work together.
- There comes a solution to this i.e. LOGICAL CLOCK.

Introduction

Solution: Timestamps to events.

If we assign the first place as 1, second place as 2, third place as 3 and so on. Then we always know that the first place will always come first and then so on.

Similarly, If we give each PC their individual number than it will be organized in a way that 1st PC will complete its process first and then second and so on.

RAMAIAH Institute of Technology

Introduction

- The concept of causality between events is fundamental to the design and analysis of parallel and distributed computing and operating systems.
- Usually causality is tracked using physical time.
- In distributed systems, it is not possible to have a global physical time.
- As asynchronous distributed computations make progress in spurts, the logical time is sufficient to capture the fundamental monotonicity property associated with causality in distributed systems.

Introduction

Three ways to implement logical clock

1. scalar time
2. vector time
3. matrix time

A framework for a system of logical clocks

- A system of logical clocks consists of a time domain T and a logical clock C. Elements of T form a partially ordered set over a relation $<$.
- Relation $<$ is called the *happened before* or *causal precedence*. Intuitively, this relation is analogous to the *earlier than* relation provided by the physical time.
- The logical clock C is a function that maps an event e in a distributed system to an element in the time domain T, denoted as $C(e)$ and called the timestamp of e, and is defined as follows:

$$C : H \mapsto T$$

such that the following property is satisfied:

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

A framework for a system of logical clocks

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

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

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

A framework for a system of logical clocks

Implementing Logical Clocks

- Implementation of logical clocks requires addressing two issues: data structures local to every process to represent logical time and a protocol to update the data structures to ensure the consistency condition.
- Each process p_i maintains data structures that allow it the following two capabilities:
  - A *local logical clock*, denoted by lc_i, that helps process p_i measure its own progress.
  - A *logical global clock*, denoted by gc_i, that is a representation of process p_i's local view of the logical global time. Typically, lc_i is a part of gc_i.

A framework for a system of logical clocks

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

- *R1*: This rule governs how the local logical clock is updated by a process when it executes an event.
- *R2*: This rule governs how a process updates its global logical clock to update its view of the global time and global progress.
- Systems of logical clocks differ in their representation of logical time and also in the protocol to update the logical clocks.

Scalar time

- Proposed by Lamport in 1978 as an attempt to totally order events in a distributed system.
- Time domain is the set of non-negative integers.
- The logical local clock of a process p_i and its local view of the global time are squashed into one integer variable C_i.
- Rules *R1* and *R2* to update the clocks are as follows:

Scalar time

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

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

In general, every time *R1* is executed, d can have a different value; however, typically d is kept at 1.

- *R2*: Each message piggybacks the clock value of its sender at sending time. When a process p_i receives a message with timestamp C_{msg}, it executes the following actions:
  - $C_i := max(C_i, C_{msg})$
  - Execute *R1*.
  - Deliver the message.

Scalar time

- Figure 3.1 shows evolution of scalar time.

P1

C_1 = 1 , 2 – send event , so piggyback the clockvalue C_{msg}

P2

C_2 = max (C_2 , C_{msg}) = max (1,2) =2

= 2+1

=3

R2: Each message piggybacks the clock value of its sender at sending time. When a process p_i receives a message with timestamp C_{msg}, it executes the following actions:

- $C_i := max(C_i, C_{msg})$
- Execute *R1*.
- Deliver the message.

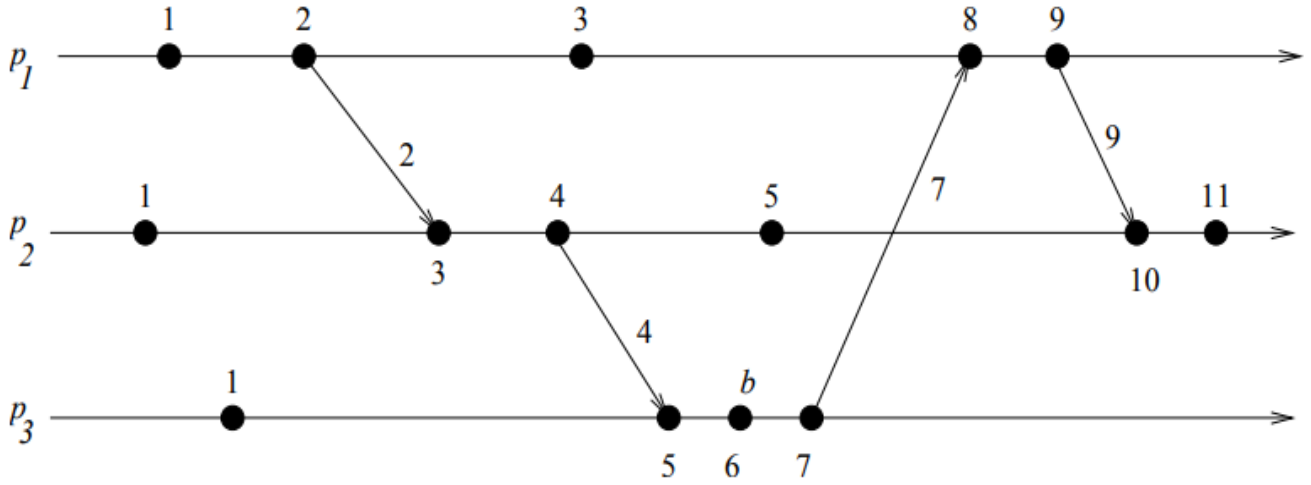


Figure 3.1: The space-time diagram of a distributed execution.

Scalar time

Basic properties

1. Consistency
2. Total Reordering
3. Event Counting
4. No strong consistency

Scalar time

Basic properties - 1.Consistency

- Scalar clocks satisfy the monotonicity and hence the consistency property:

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

Scalar time

Basic properties -2.Total Reordering

- Scalar clocks can be used to totally order events in a distributed system.
- The main problem in totally ordering events is that two or more events at different processes may have identical timestamp.
- For example in Figure 3.1, the third event of process P_1 and the second event of process P_2 have identical scalar timestamp.

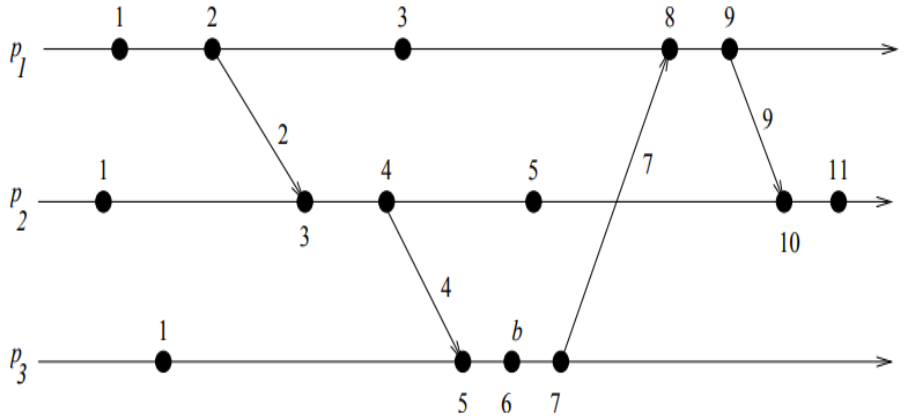


Figure 3.1: The space-time diagram of a distributed execution.

Scalar time

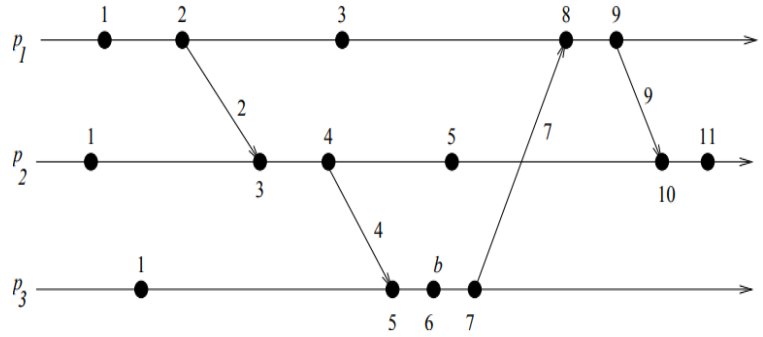


Figure 3.1: The space-time diagram of a distributed execution.

Basic properties -2.Total Reordering

A tie-breaking mechanism is needed to order such events. A tie is broken as follows:

- Process identifiers are linearly ordered and tie among events with identical scalar timestamp is broken on the basis of their process identifiers.
- The lower the process identifier in the ranking, the higher the priority.
- The timestamp of an event is denoted by a tuple (t, i) where t is its time of occurrence and i is the identity of the process where it occurred.
- The total order relation $\prec$ on two events x and y with timestamps (h,i) and (k,j), respectively, is defined as follows:

$$x \prec y \Leftrightarrow (h < k \textbf{ or } (h = k \textbf{ and } i < j))$$

(T X, P1) = 3,1

(T Y, P2) = 3,2

Scalar time

Basic properties -3.Event Counting

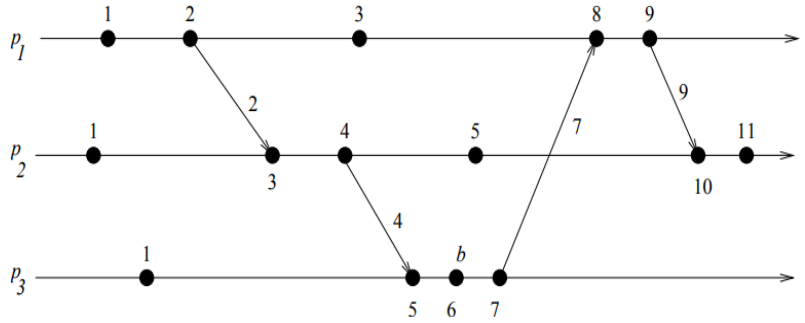


Figure 3.1: The space-time diagram of a distributed execution.

- If the increment value *d* is always 1, the scalar time has the following interesting property: if event *e* has a timestamp *h*, then *h-1* represents the minimum logical duration, counted in units of events, required before producing the event *e*;
- We call it the height of the event *e*.
- In other words, *h-1* events have been produced sequentially before the event *e* regardless of the processes that produced these events.
- For example, in Figure 3.1, five events precede event *b* on the longest causal path ending at *b*.

Scalar time

Basic properties -4.No strong consistency

- The system of scalar clocks is not strongly consistent; that is, for two events e_i and e_j, $C(e_i) < C(e_j) \not\Longrightarrow e_i \rightarrow e_j$.
- For example, in Figure 3.1, the third event of process P_1 has smaller scalar timestamp than the third event of process P_2.However, the former did not happen before the latter.

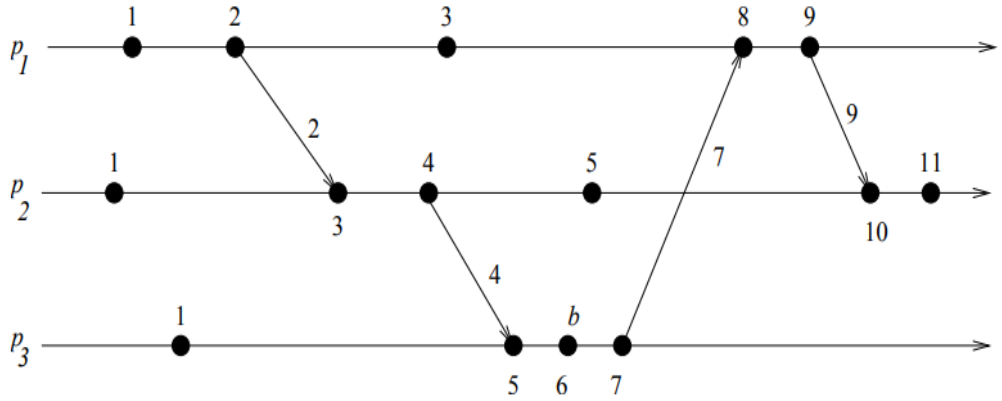


Figure 3.1: The space-time diagram of a distributed execution.

Scalar time

Basic properties -4.No strong consistency

- The reason that scalar clocks are not strongly consistent is that the logical local clock and logical global clock of a process are squashed into one, resulting in the loss causal dependency information among events at different processes.
- For example, in Figure 3.1, when process P_2 receives the first message from process P_1, it updates its clock to 3, forgetting that the timestamp of the latest event at P_1 on which it depends is 2.

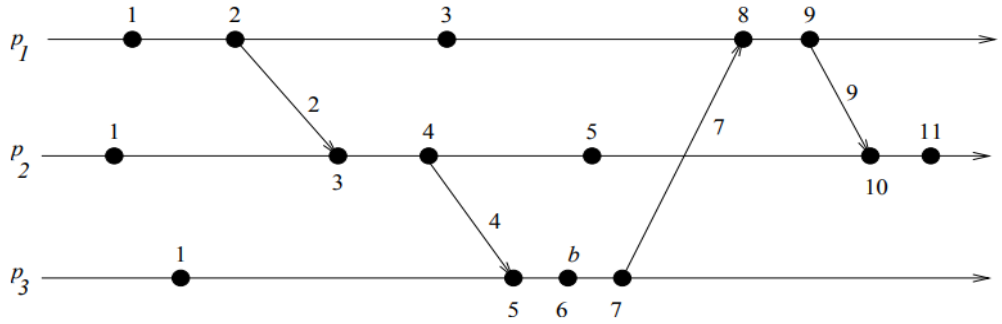


Figure 3.1: The space-time diagram of a distributed execution.

Vector time

- **Vector Clock** is an algorithm that generates partial ordering of events and detects causality violations in a distributed system.
- This algorithm helps us label every process with a vector(a list of integers) with an integer for each local clock of every process within the system.
- So for N given processes, there will be vector/ array of size N.

Vector time

How does the vector clock algorithm work :

- Initially, all the clocks are set to zero.
- Every time, an Internal event occurs in a process, the value of the processes's logical clock in the vector is incremented by 1
- Also, every time a process sends a message, the value of the processes's logical clock in the vector is incremented by 1.
- Every time, a process receives a message, the value of the processes's logical clock in the vector is incremented by 1, and moreover, each element is updated by taking the maximum of the value in its own vector clock and the value in the vector in the received message (for every element).

Vector time

- The system of vector clocks was developed independently by Fidge, Mattern and Schmuck.
- In the system of vector clocks, the time domain is represented by a set of n-dimensional non-negative integer vectors.
- Each process p_i maintains a vector $vt_i[1..n]$, where $vt_i[i]$ is the local logical clock of p_i and describes the logical time progress at process p_i.
- $vt_i[j]$ represents process p_i's latest knowledge of process p_j local time.
- If $vt_i[j]=x$, then process p_i knows that local time at process p_j has progressed till x.
- The entire vector vt_i constitutes p_i's view of the global logical time and is used to timestamp events.

Vector time

Process p_i uses the following two rules *R1* and *R2* to update its clock:

- *R1*: Before executing an event, process p_i updates its local logical time as follows:

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

- *R2*: Each message m is piggybacked with the vector clock vt of the sender process at sending time. On the receipt of such a message (m, vt), process p_i executes the following sequence of actions:
  - Update its global logical time as follows:

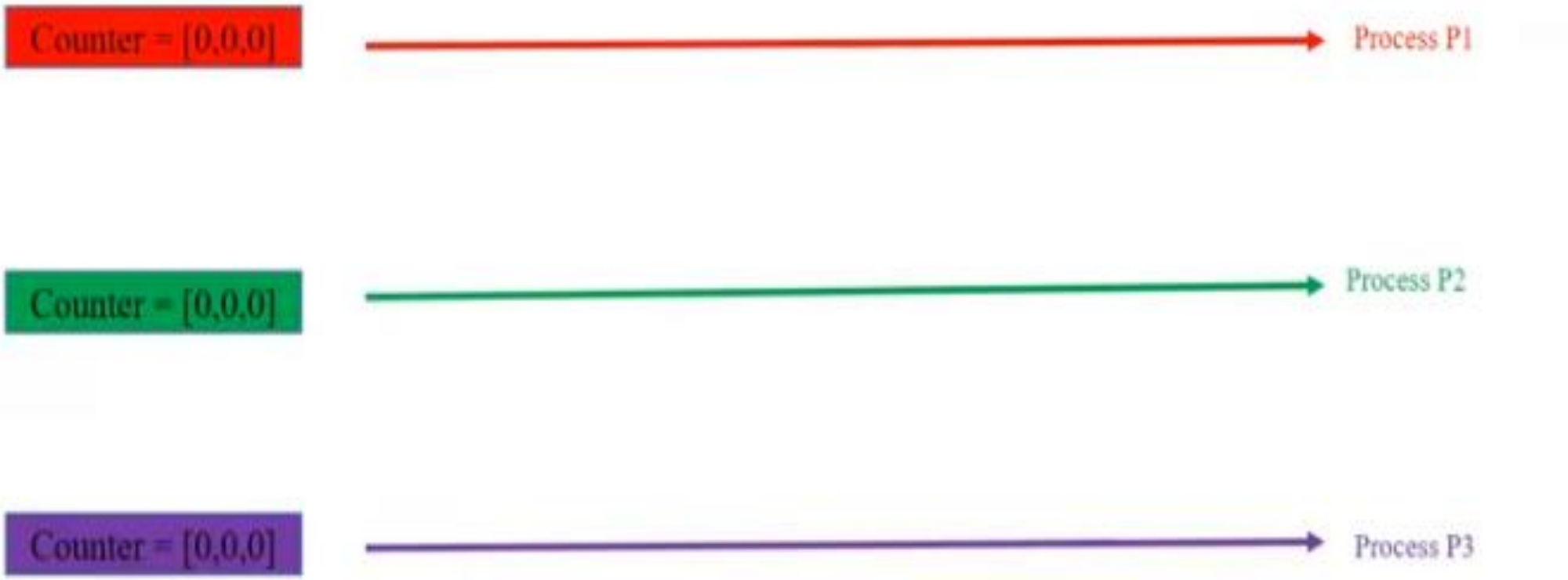
$$1 \leq k \leq n \ : \ vt_i[k] := max(vt_i[k], vt[k])$$

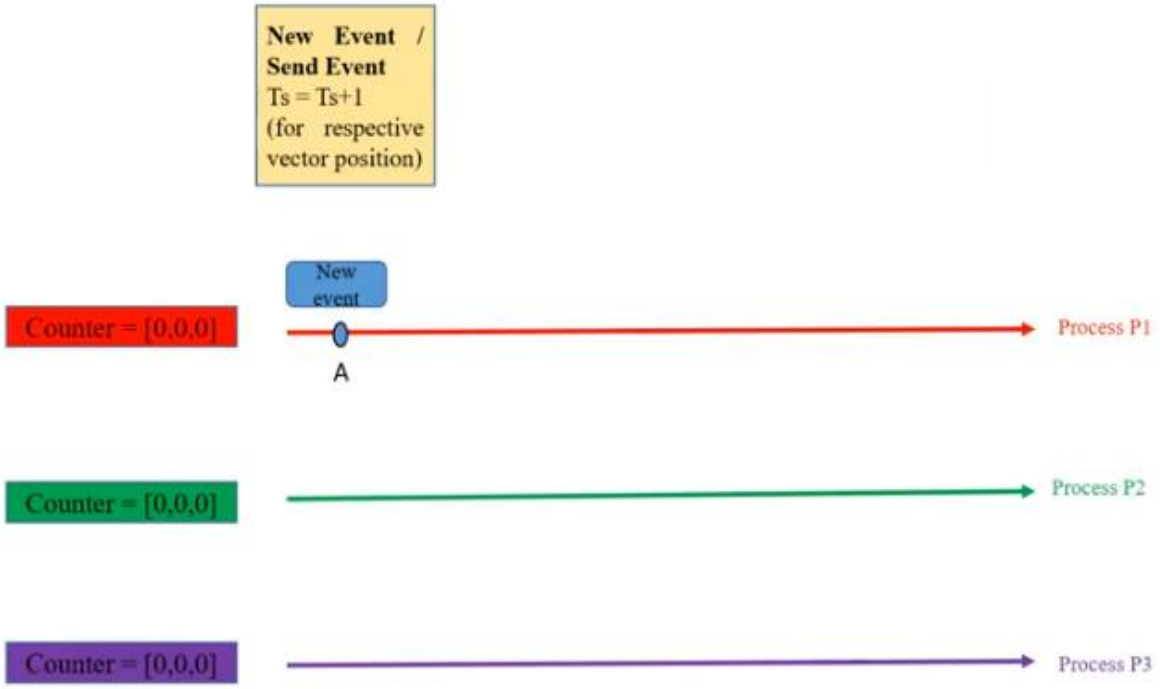
  - Execute *R1*.
  - Deliver the message m.

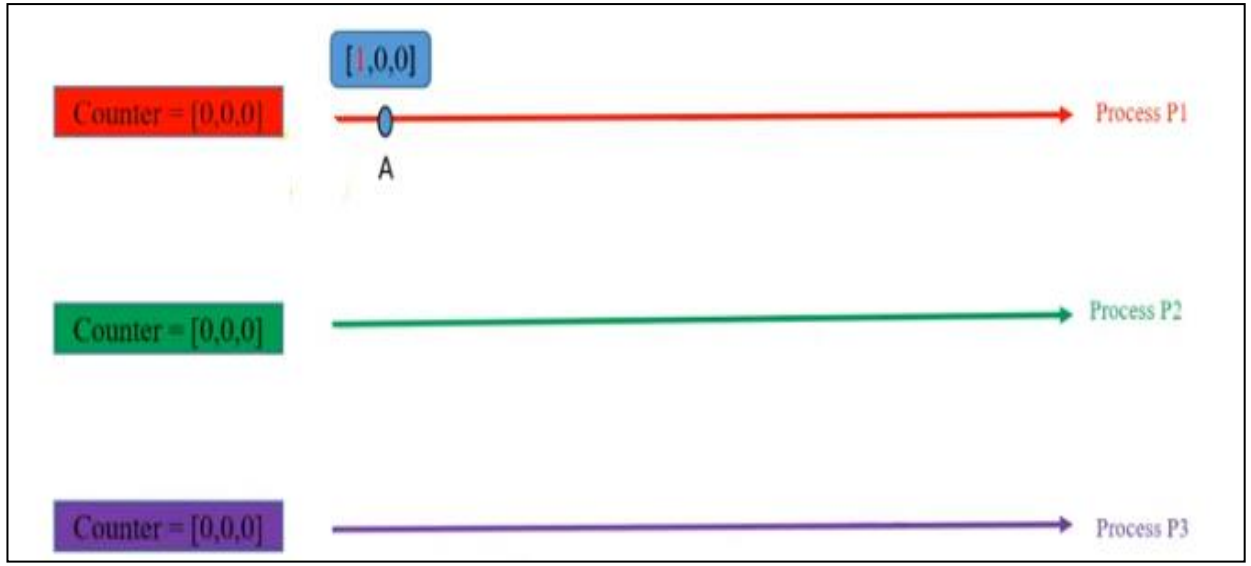
Vector time

How does the vector clock algorithm work :

- Initially, all the clocks are set to zero.
- Every time, an Internal event occurs in a process, the value of the processes's logical clock in the vector is incremented by 1
- Also, every time a process sends a message, the value of the processes's logical clock in the vector is incremented by 1.
- Every time, a process receives a message, the value of the processes's logical clock in the vector is incremented by 1, and moreover, each element is updated by taking the maximum of the value in its own vector clock and the value in the vector in the received message (for every element).


Counter = [0,0,0]
Process P1
Counter = [0,0,0]
Process P2
Counter = [0,0,0]
Process P3

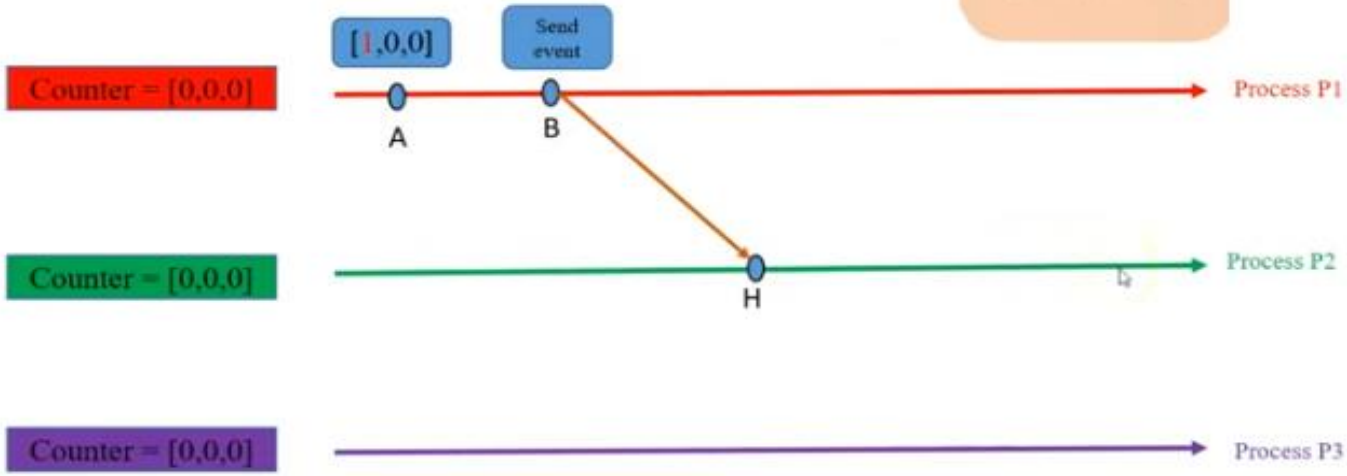

New Event / Send Event
Ts = Ts+1
(for respective vector position)
New event
Counter = [0,0,0]
Process P1
A
Counter = [0,0,0]
Process P2
Counter = [0,0,0]
Process P3

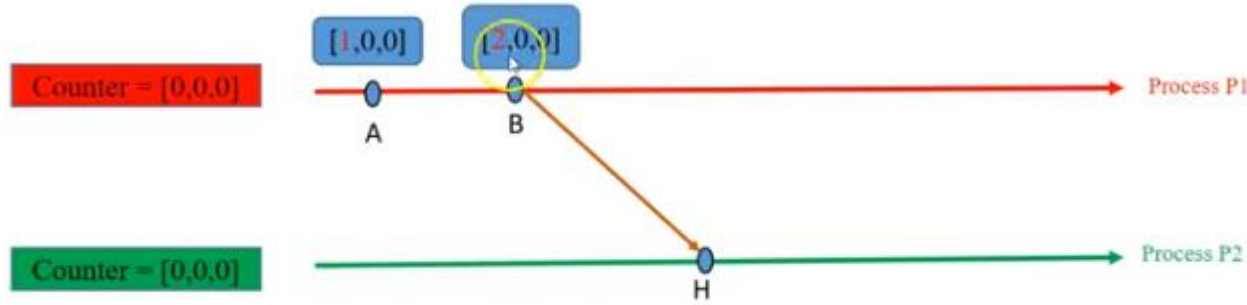

[1,0,0]
Counter = [0,0,0]
Process P1
A
Counter = [0,0,0]
Process P2
Counter = [0,0,0]
Process P3

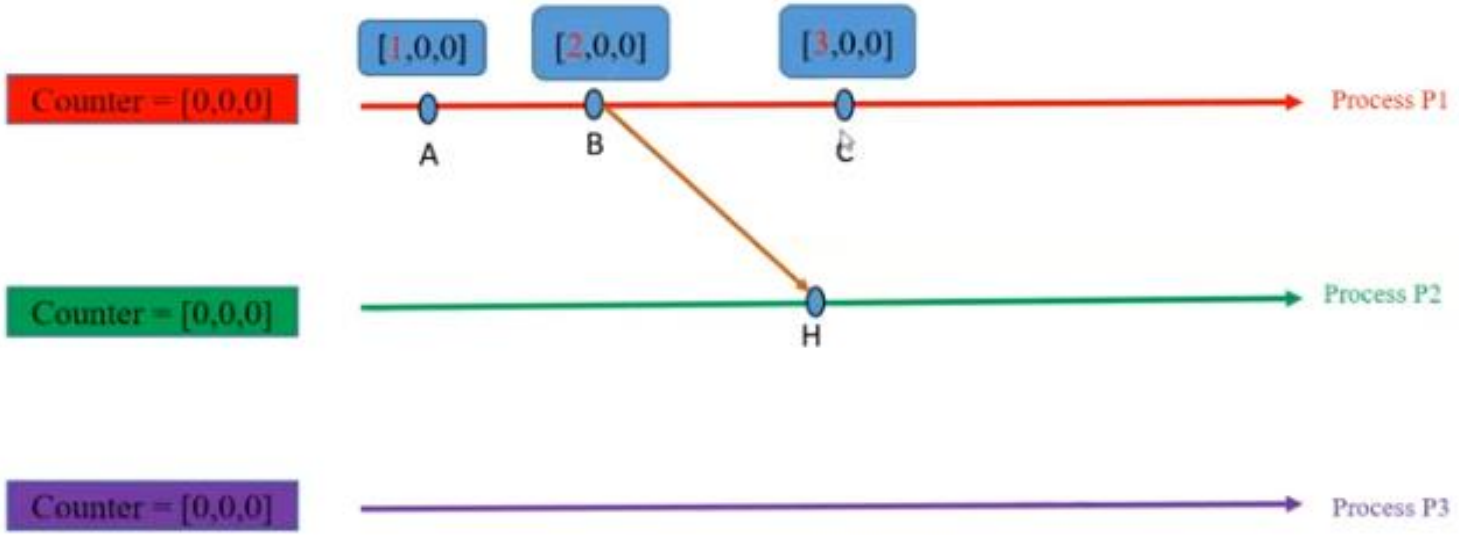
New Event / Send Event
Ts = Ts+1
(for respective vector position)

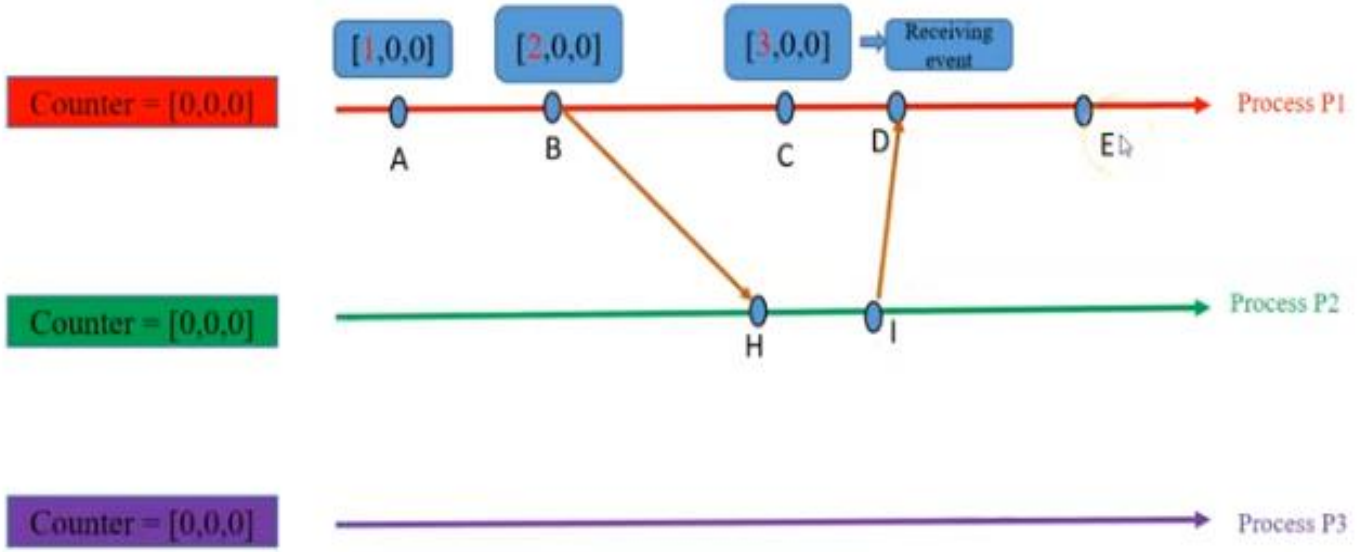
Receiving event

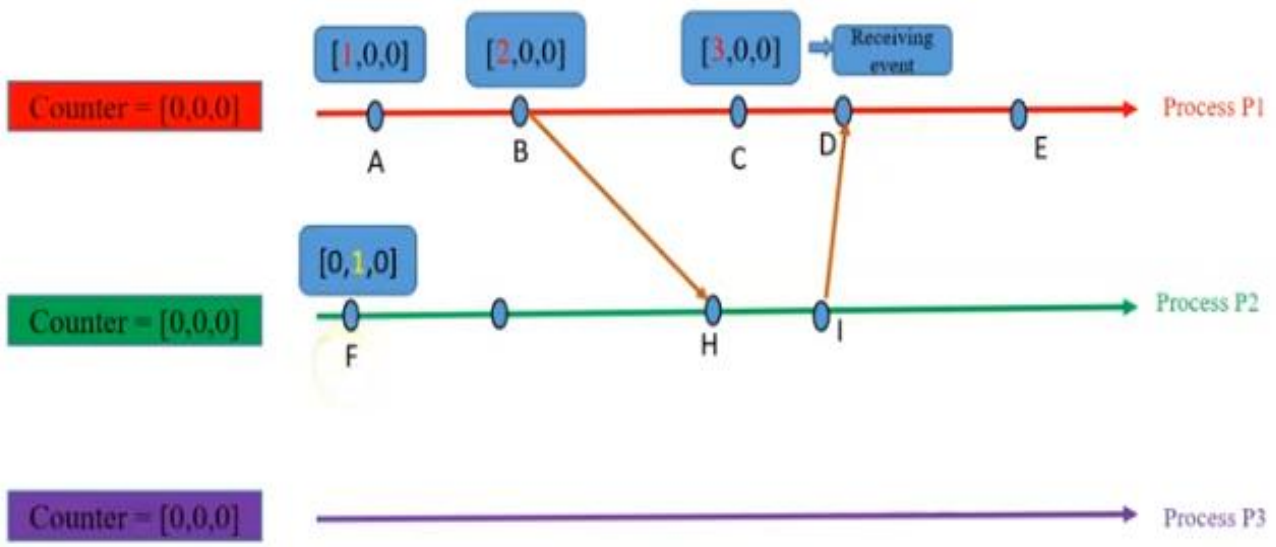
- Compare its own vector and the senders vector element position wise
- **New TS** = max (own ts, sender ts)

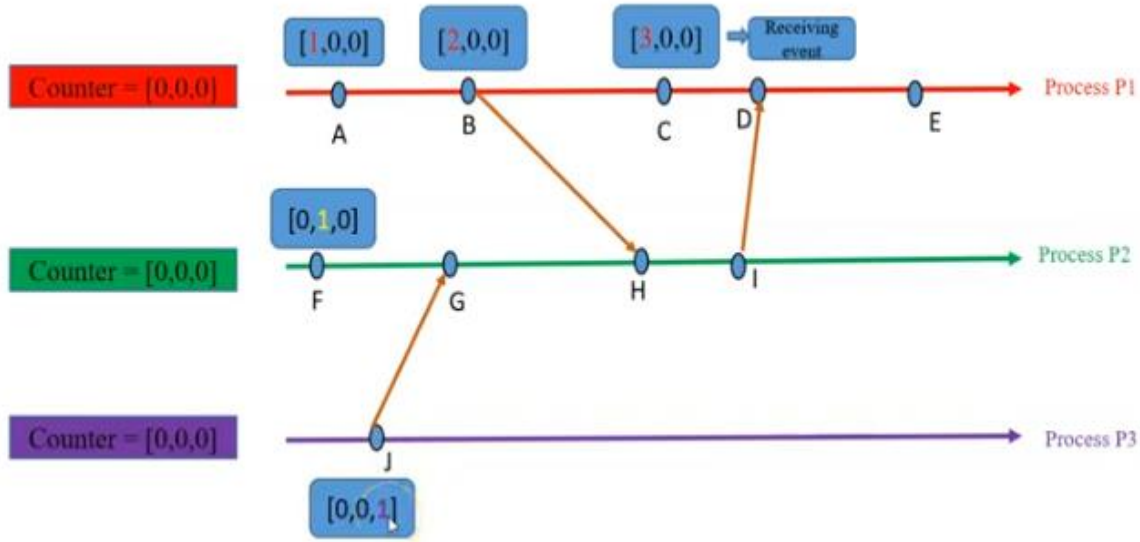





[1,0,0]
[2,0,0]
[3,0,0]
Counter = [0,0,0]
Process P1
A
B
C
Counter = [0,0,0]
Process P2
H
Counter = [0,0,0]
Process P3


[1,0,0]
[2,0,0]
[3,0,0]
Receiving event
Counter = [0,0,0]
Process P1
A
B
C
D
E
Counter = [0,0,0]
Process P2
H
I
Counter = [0,0,0]
Process P3

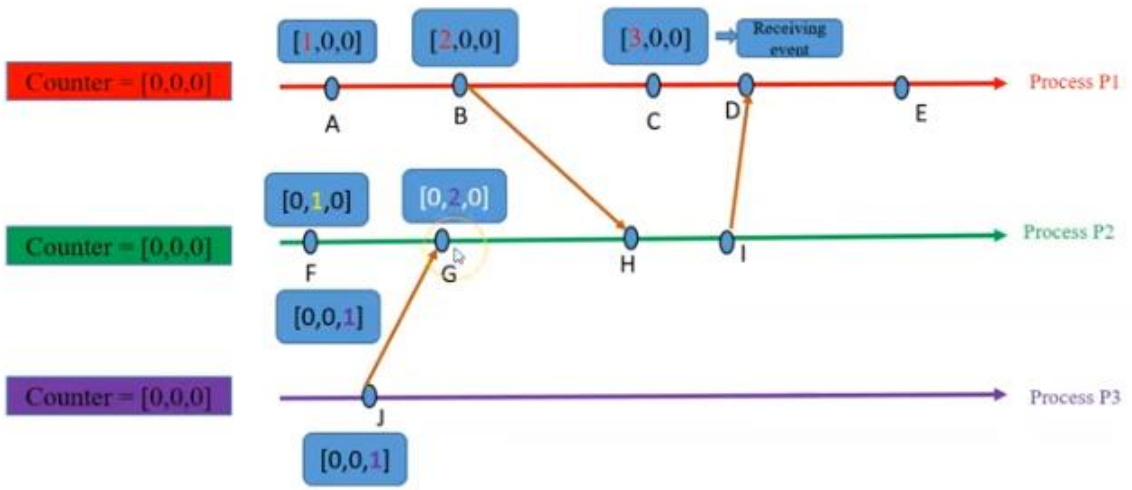

[1,0,0]
[2,0,0]
[3,0,0]
Receiving event
Counter = [0,0,0]
Process P1
A
B
C
D
E
[0,1,0]
Counter = [0,0,0]
Process P2
F
H
I
Counter = [0,0,0]
Process P3

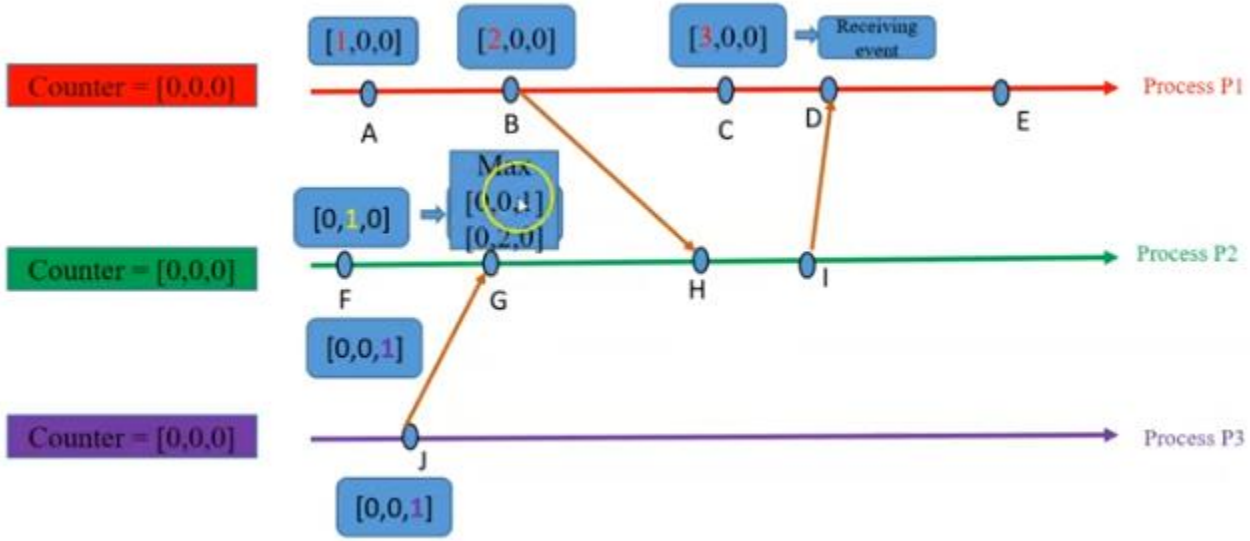

[1,0,0]
[2,0,0]
[3,0,0]
Receiving event
Counter = [0,0,0]
Process P1
A
B
C
D
E
[0,1,0]
Counter = [0,0,0]
Process P2
F
G
H
I
Counter = [0,0,0]
Process P3
J
[0,0,1]

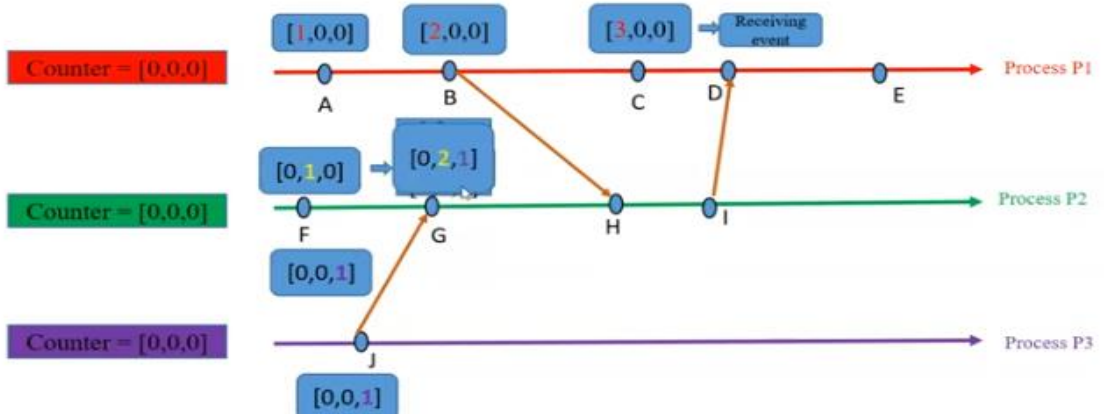
New Event / Send Event
Ts = Ts+1
(for respective vector position)

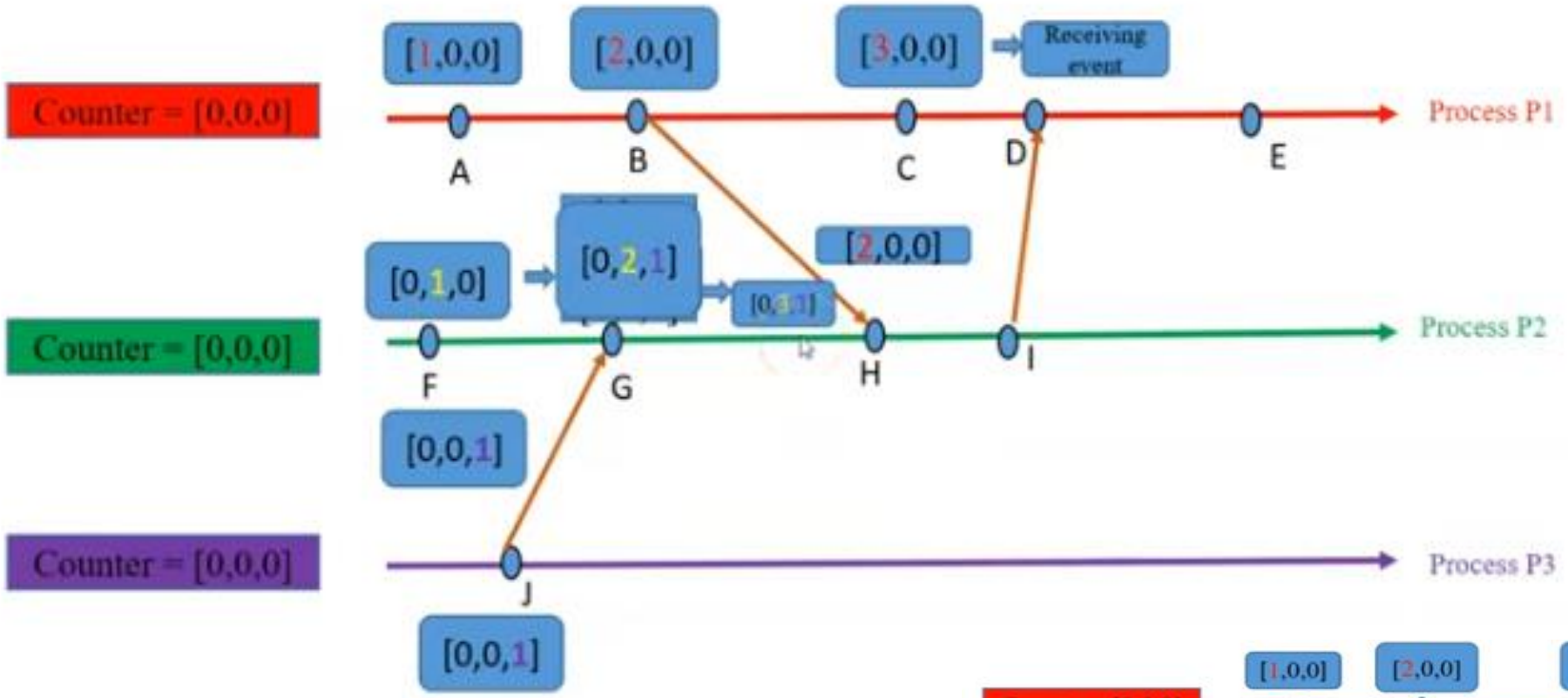
Receiving event

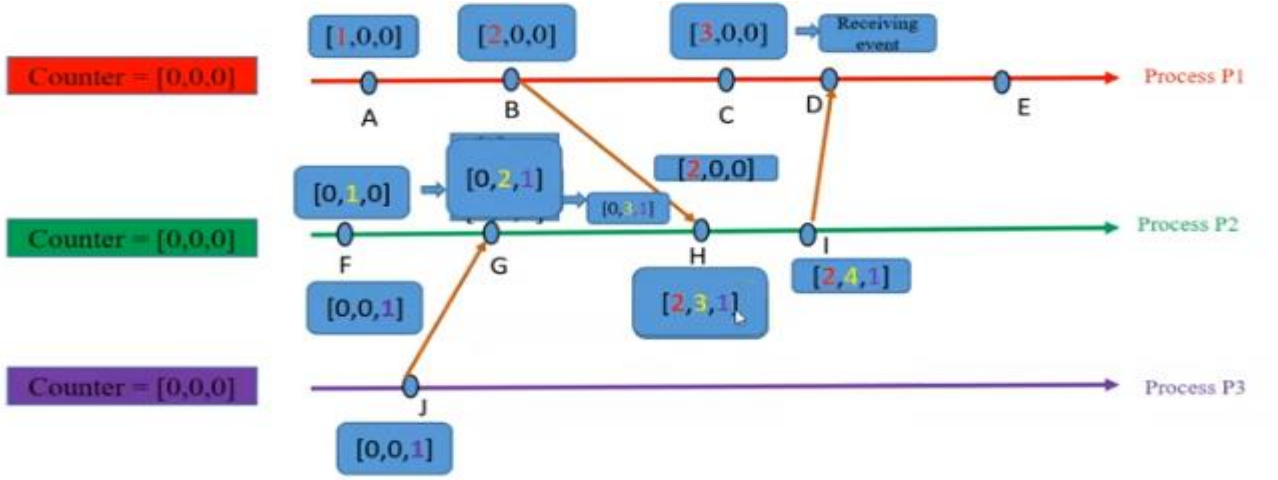
- Compare its own vector and the senders vector element position wise
- **New TS** = max (own ts, sender ts)







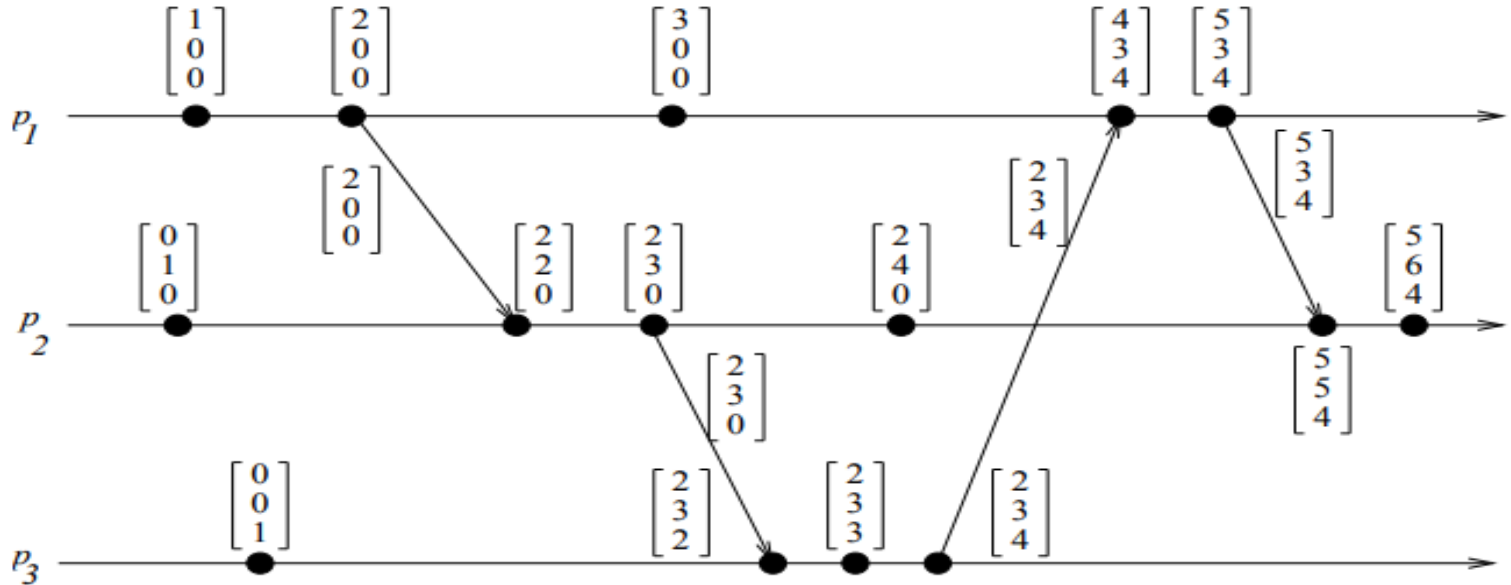

[1,0,0]
[2,0,0]
[3,0,0]
Receiving event
Counter = [0,0,0]
Process P1
A
B
C
D
E
[0,1,0]
[0,2,1]
[2,0,0]
[0,3,1]
Counter = [0,0,0]
Process P2
F
G
H
I
[0,0,1]
Counter = [0,0,0]
Process P3
J
[0,0,1]


[1,0,0]
[2,0,0]
[3,0,0]
Receiving event
Counter = [0,0,0]
Process P1
A
B
C
D
E
[0,1,0]
[0,2,1]
[2,0,0]
[0,3,1]
Counter = [0,0,0]
Process P2
F
G
H
I
[2,4,1]
[0,0,1]
[2,3,1]
Counter = [0,0,0]
Process P3
J
[0,0,1]

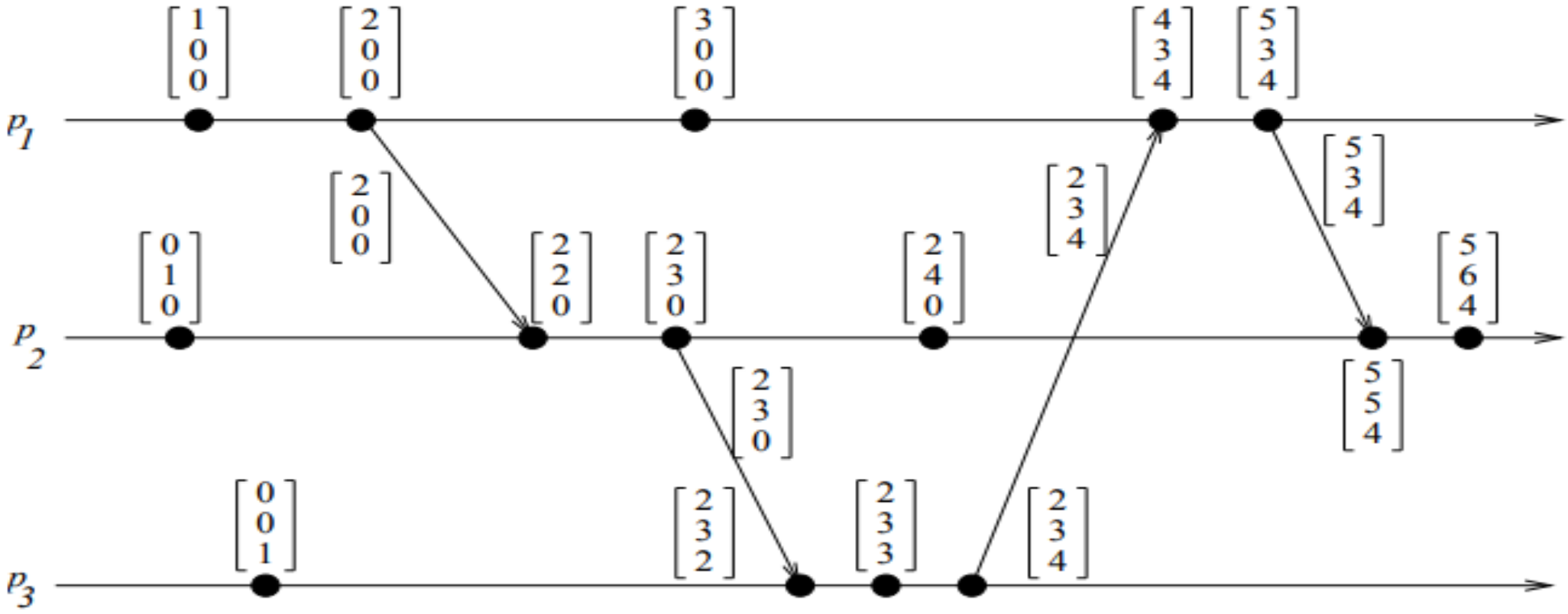
RAMAIAH
Institute of Technology

Vector time

- The timestamp of an event is the value of the vector clock of its process when the event is executed.
- Figure shows an example of vector clocks progress with the increment value *d=1*.
- Initially, a vector clock is $[0, 0, 0, ..., 0]$.



Vector time


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

Vector time-Comparing Vector Timestamps

- The following relations are defined to compare two vector timestamps, vh and vk:

$$
\begin{aligned}
vh = vk &\Leftrightarrow \forall x : vh[x] = vk[x] \\
vh \leq vk &\Leftrightarrow \forall x : vh[x] \leq vk[x] \\
vh < vk &\Leftrightarrow vh \leq vk \text{ and } \exists x : vh[x] < vk[x] \\
vh \parallel vk &\Leftrightarrow \neg(vh < vk) \wedge \neg(vk < vh)
\end{aligned}
$$

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

$$
\begin{aligned}
x \rightarrow y &\Leftrightarrow vh[i] \leq vk[i] \\
x \parallel y &\Leftrightarrow vh[i] > vk[i] \wedge vh[j] < vk[j]
\end{aligned}
$$

Vector time – Basic Properties

Isomorphism

- If events in a distributed system are timestamped using a system of vector clocks, we have the following property.
  If two events x and y have timestamps vh and vk, respectively, then

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

- Thus, there is an isomorphism between the set of partially ordered events produced by a distributed computation and their vector timestamps.

Vector time – Basic Properties

Strong Consistency

- The system of vector clocks is strongly consistent; thus, by examining the vector timestamp of two events, we can determine if the events are causally related.
- However, Charron-Bost showed that the dimension of vector clocks cannot be less than *n*, the total number of processes in the distributed computation, for this property to hold.

Vector time – Basic Properties

Event Counting

- If d=1 (in rule *R1*), then the i^{th} component of vector clock at process p_i, $vt_i[i]$, denotes the number of events that have occurred at p_i until that instant.
- So, if an event e has timestamp vh, $vh[j]$ denotes the number of events executed by process p_j that causally precede e. Clearly, $\sum vh[j] - 1$ represents the total number of events that causally precede e in the distributed computation.

Vector time – Applications

Since vector time tracks causal dependencies

1. Used in distributed debugging,
2. Implementations of causal ordering communication and causal distributed shared memory,
3. Establishment of global breakpoints,
4. In determining the consistency of checkpoints in optimistic recovery.

UNIT -1

Logical time: (Chapter 3: 3.1,3.2,3.3,3.4,3.5,3.6)

- Introduction
- A framework for a system of logical clocks
- Scalar time
- Vector time
- efficient implementations of vector clocks
- Jard–Jourdan's adaptive technique

Efficient implementations of vector clocks

- If the number of processes in a distributed computation is large, then vector clocks will require piggybacking of huge amount of information in messages for the purpose of disseminating time progress and updating clocks.
- The message overhead grows linearly with the number of processors in the system and when there are thousands of processors in the system, the message size becomes huge even if there are only a few events occurring in few processors.

So efficient ways to maintain vector clocks is required

Efficient implementations of vector clocks

Different ways to maintain vector clocks are

1. Singhal–Kshemkalyani's differential technique
2. Fowler–Zwaenepoel's direct-dependency technique
3. Jard–Jourdan's adaptive technique

Efficient implementations of vector clocks

Singhal–Kshemkalyani's differential technique

- *Singhal-Kshemkalyani's differential technique* is based on the observation that between successive message sends to the same process, only a few entries of the vector clock at the sender process are likely to change.
- When a process p_i sends a message to a process p_j, it piggybacks only those entries of its vector clock that differ since the last message sent to p_j.
- If entries $i_1, i_2, \ldots, i_{n_1}$ of the vector clock at p_i have changed to $v_1, v_2, \ldots, v_{n_1}$, respectively, since the last message sent to p_j, then process p_i piggybacks a compressed timestamp of the form:

$$\{(i_1, v_1), (i_2, v_2), \ldots, (i_{n_1}, v_{n_1})\}$$

to the next message to p_j.

Efficient implementations of vector clocks

Singhal–Kshemkalyani's differential technique

When p_j receives this message, it updates its vector clock as follows:

$$vt_i[i_k] = max(vt_i[i_k], v_k) \text{ for } k = 1, 2, \ldots, n_1.$$

- Thus this technique cuts down the message size, communication bandwidth and buffer (to store messages) requirements.
- In the worst of case, every element of the vector clock has been updated at p_i since the last message to process p_j, and the next message from p_i to p_j will need to carry the entire vector timestamp of size n.
- However, on the average the size of the timestamp on a message will be less than n.

Efficient implementations of vector clocks

Singhal–Kshemkalyani's differential technique

- Implementation of this technique requires each process to remember the vector timestamp in the message last sent to every other process.
- Direct implementation of this will result in $O(n^2)$ storage overhead at each process.
- Singhal and Kshemkalyani developed a clever technique that cuts down this storage overhead at each process to $O(n)$.

Efficient implementations of vector clocks

Singhal–Kshemkalyani's differential technique

The technique works in the following manner:

- Process p_i maintains the following two additional vectors:
  - $LS_i[1..n]$ ('Last Sent'):
    $LS_i[j]$ indicates the value of $vt_i[i]$ when process p_i last sent a message to process p_j.
  - $LU_i[1..n]$ ('Last Update'):
    $LU_i[j]$ indicates the value of $vt_i[i]$ when process p_i last updated the entry $vt_i[j]$.
- Clearly, $LU_i[i] = vt_i[i]$ at all times and $LU_i[j]$ needs to be updated only when the receipt of a message causes p_i to update entry $vt_i[j]$. Also, $LS_i[j]$ needs to be updated only when p_i sends a message to p_j.

Efficient implementations of vector clocks

Singhal–Kshemkalyani's differential technique

- Since the last communication from p_i to p_j, only those elements of vector clock $vt_i[k]$ have changed for which $LS_i[j] < LU_i[k]$ holds.
- Hence, only these elements need to be sent in a message from p_i to p_j. When p_i sends a message to p_j, it sends only a set of tuples

$$\{(x, vt_i[x]) | LS_i[j] < LU_i[x]\}$$

as the vector timestamp to p_j, instead of sending a vector of n entries in a message.

Efficient implementations of vector clocks

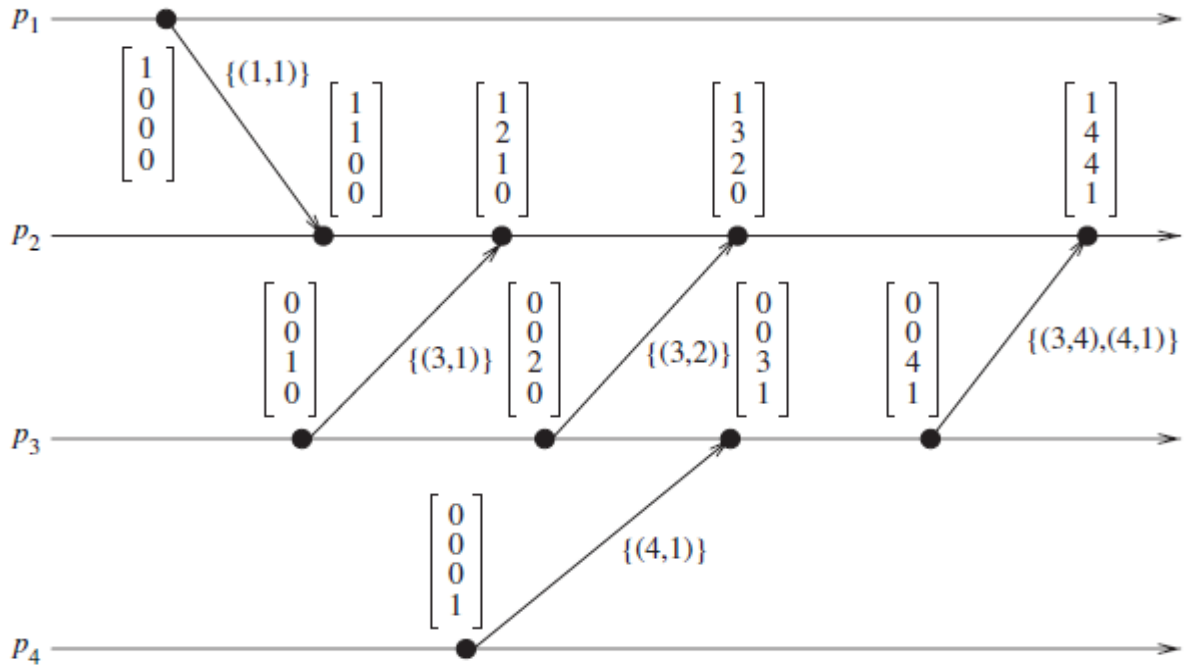
Singhal–Kshemkalyani's differential technique

- Thus the entire vector of size n is not sent along with a message. Instead, only the elements in the vector clock that have changed since the last message send to that process are sent in the format $\{(p_1, latest\_value), (p_2, latest\_value), \ldots\}$, where p_i indicates that the p_ith component of the vector clock has changed.
- This technique requires that the communication channels follow FIFO discipline for message delivery.

Efficient implementations of vector clocks

Singhal–Kshemkalyani's differential technique

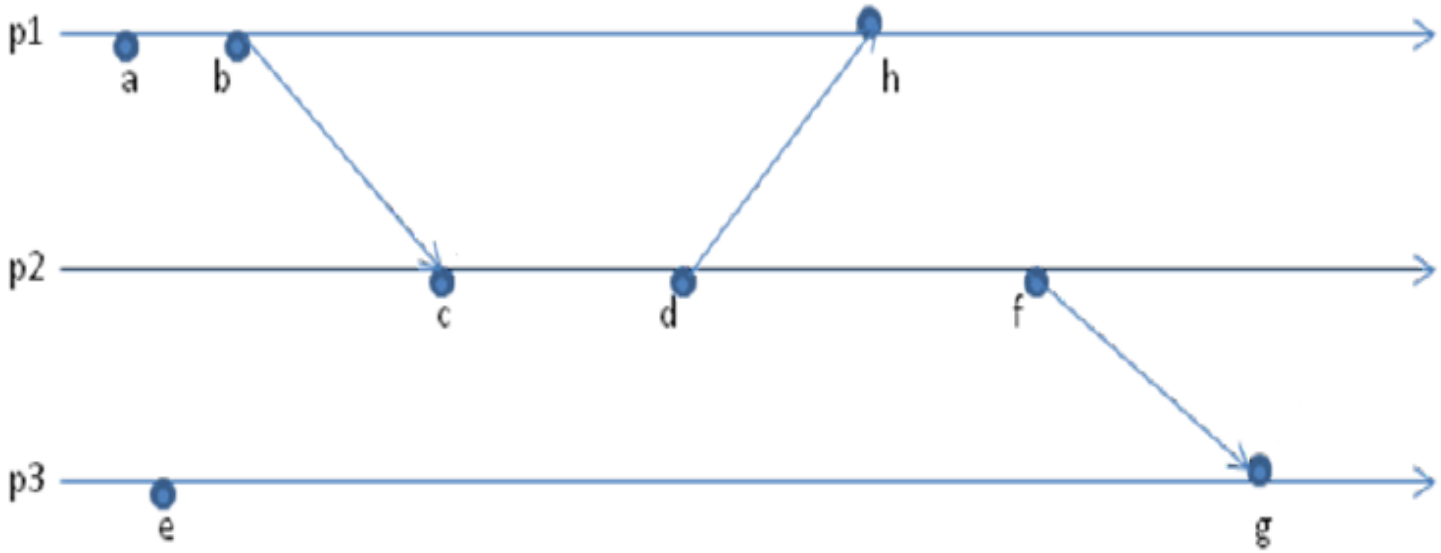
For instance, the second message from p_3 to p_2 (which contains a timestamp $\{(3,2)\}$) informs p_2 that the third component of the vector clock has been modified and the new value is 2.



Efficient implementations of vector clocks

Singhal–Kshemkalyani's differential technique

Q1: Apply Singhal-Kshemkalyani's differential technique for the given space timing diagram, and compute the vector clock progress.

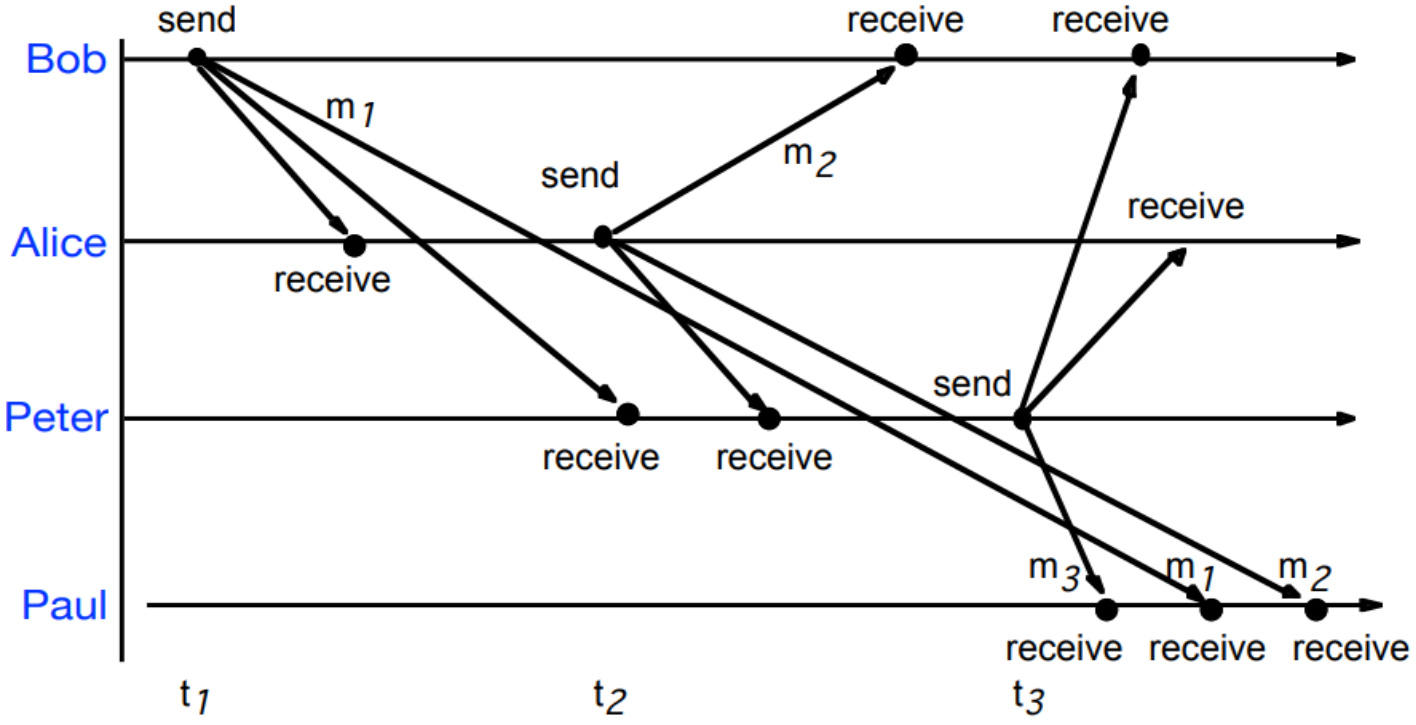


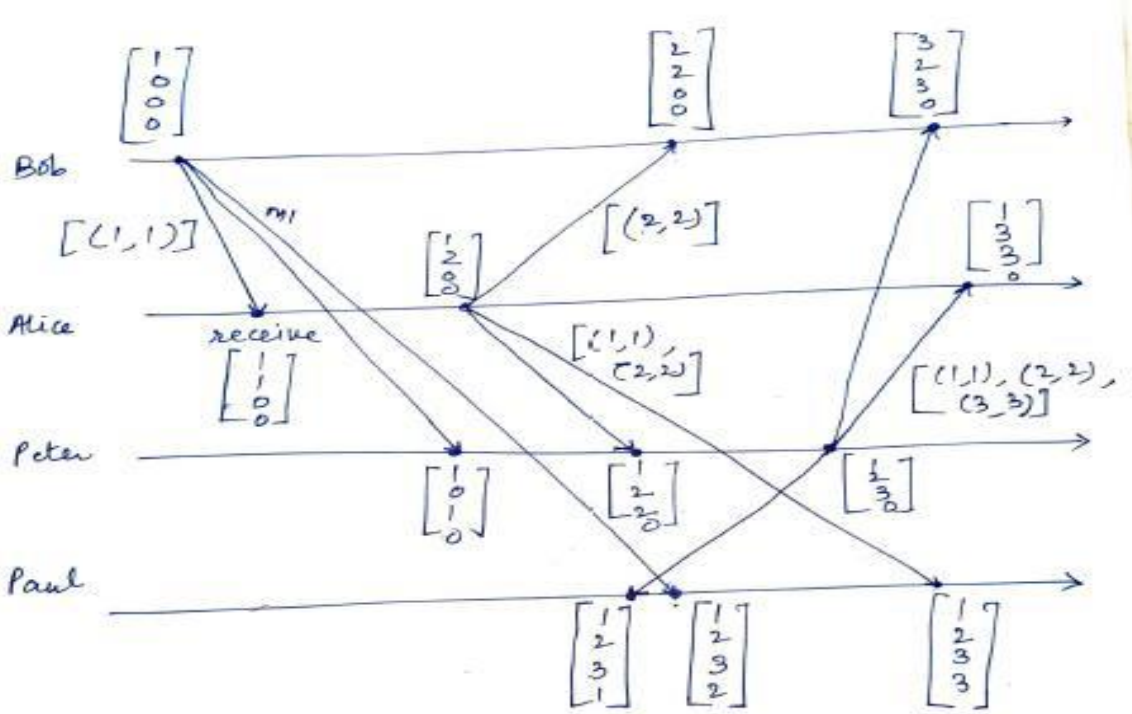
RAMAIAH
Institute of Technology

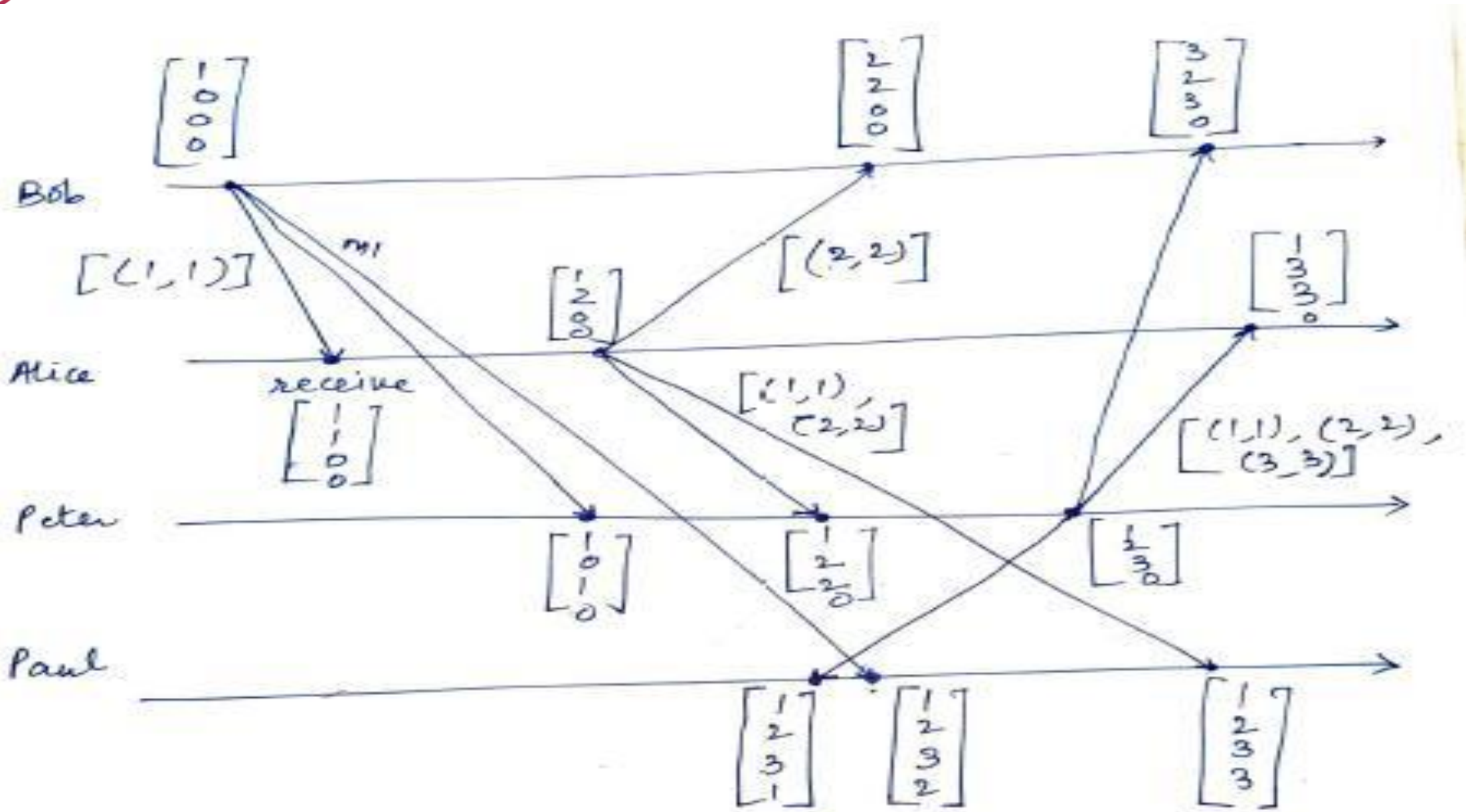
Efficient implementations of vector clocks

Singhal–Kshemkalyani's differential technique

Q2: Apply Singhal-Kshemkalyani's differential technique for the given space timing diagram, and compute the vector clock progress.






Bob
Alice
Peter
Paul
[(1,1)]
receive
[(2,2)]
[(1,1),
(2,2)]
[(1,1), (2,2),
(3,3)]

Efficient implementations of vector clocks

Fowler–Zwaenepoel's direct-dependency technique

- Fowler–Zwaenepoel direct dependency technique reduces the size of messages by transmitting only a scalar value in the messages.
- No vector clocks are maintained on-the-fly.
- Instead, a process only maintains information regarding direct dependencies on other processes.
- A vector time for an event, which represents transitive dependencies on other processes, is constructed off-line from a recursive search of the direct dependency information at processes.

Efficient implementations of vector clocks

Fowler–Zwaenepoel's direct-dependency technique

Each process p_i maintains a dependency vector D_i. Initially,

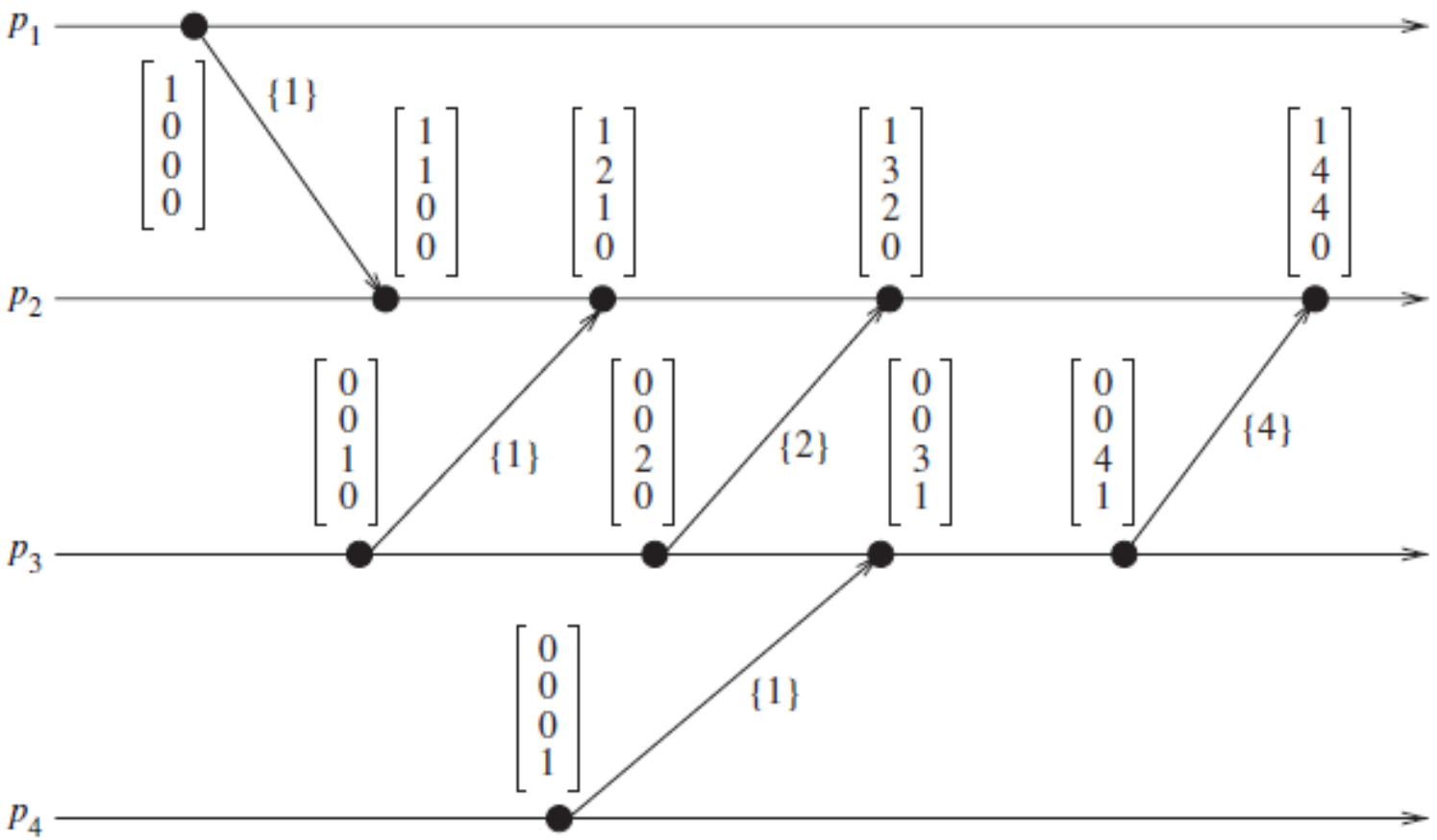
$$D_i[j] = 0 \text{ for } j = 1, \ldots, n.$$

D_i is updated as follows:

1. Whenever an event occurs at p_i, $D_i[i] := D_i[i] + 1$. That is, the vector component corresponding to its own local time is incremented by one.
2. When a process p_i sends a message to process p_j, it piggybacks the updated value of $D_i[i]$ in the message.
3. When p_i receives a message from p_j with piggybacked value d, p_i updates its dependency vector as follows: $D_i[j] := max\{D_i[j], d\}$.

Efficient implementations of vector clocks

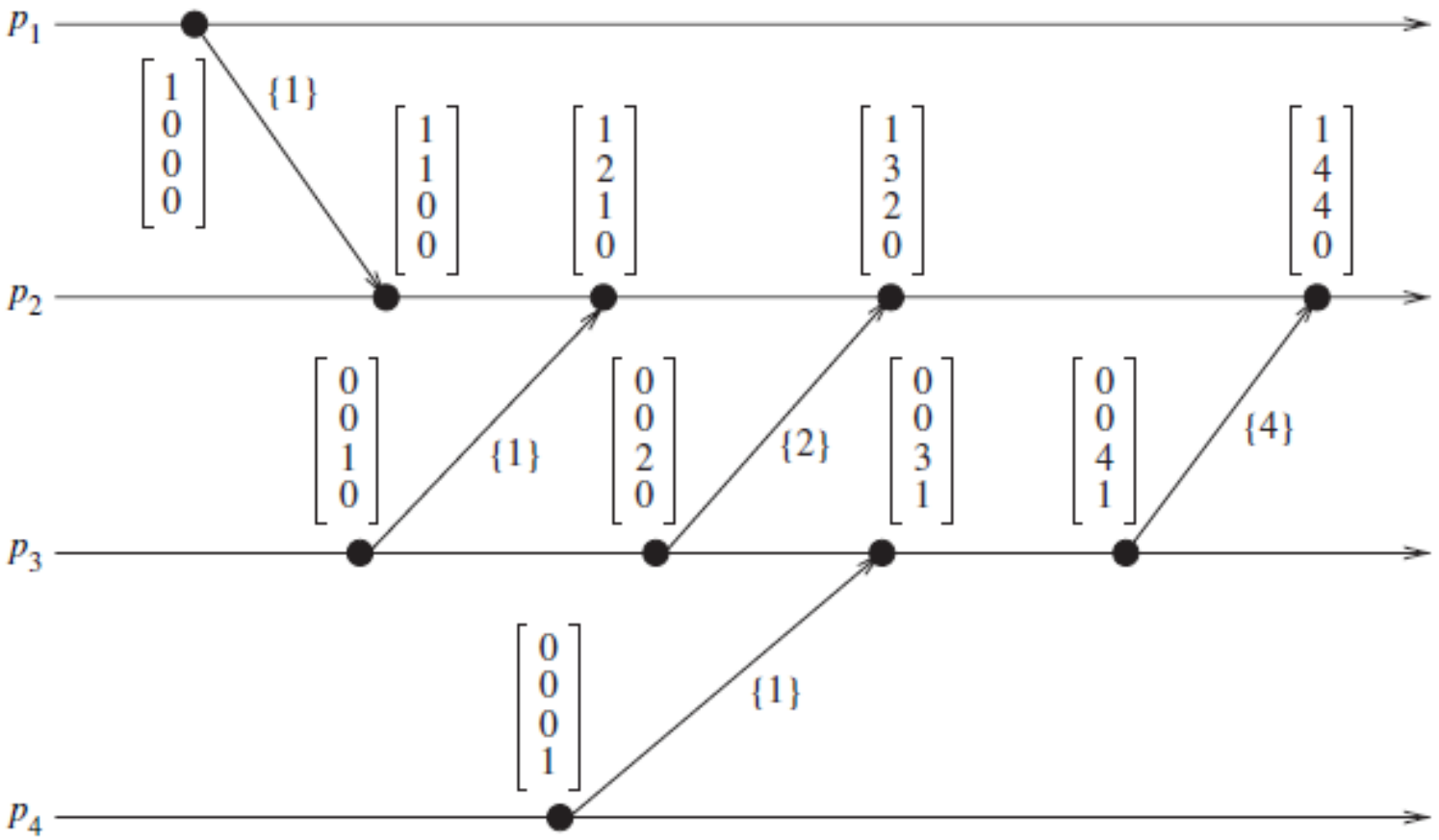
When process p4 sends a message to process p3, it piggybacks a scalar that indicates the direct dependency of p3 on p4 because of this message.



Efficient implementations of vector clocks

Subsequently,

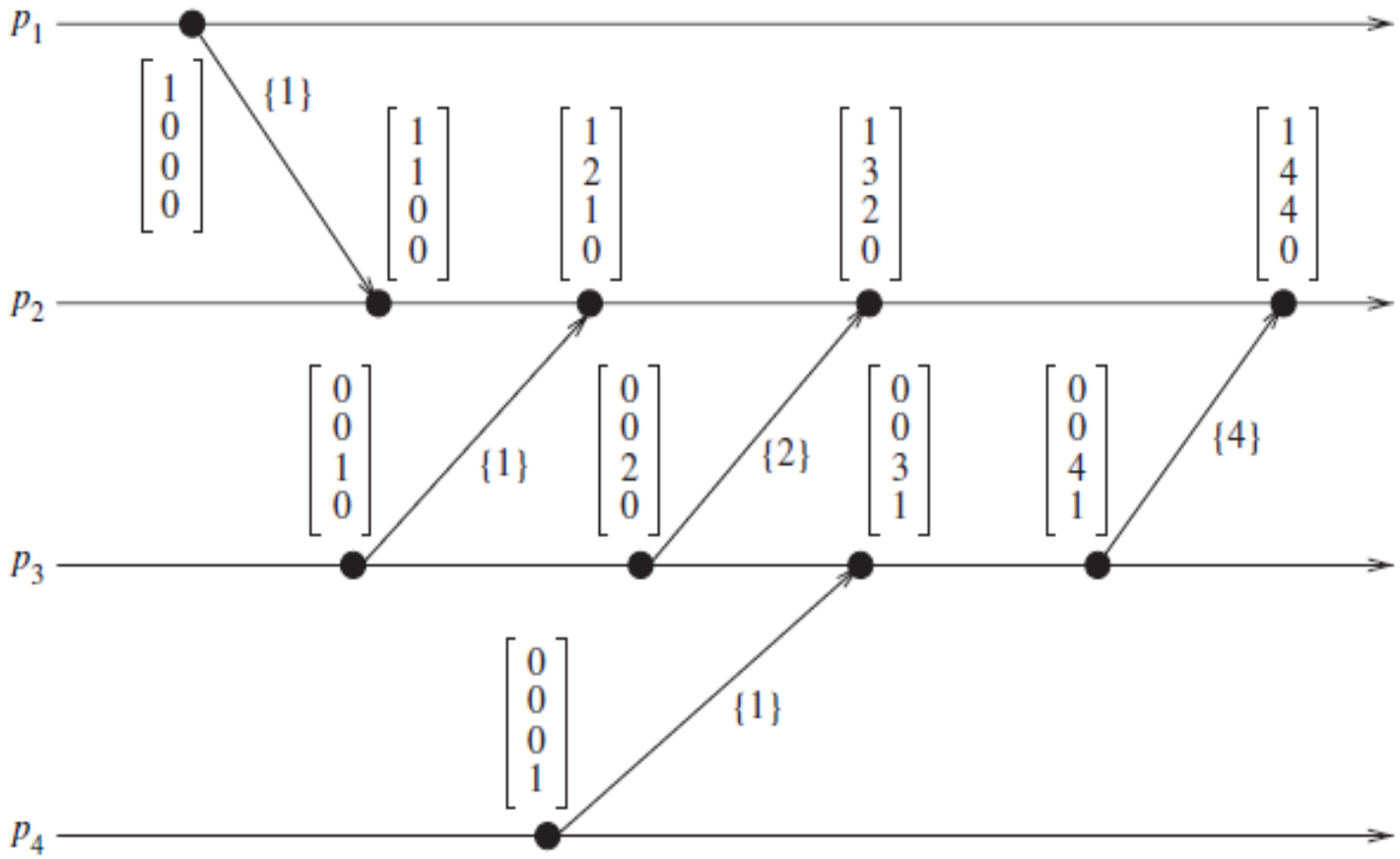
process p3 sends a message to process p2 piggybacking a scalar to indicate the direct dependency of p2 on p3 because of this message.



Efficient implementations of vector clocks

Now, process p2 is in fact indirectly dependent on process p4 since process p3 is dependent on process p4.

However, process p2 is never informed about its indirect dependency on p4..



Efficient implementations of vector clocks

Drawback

- The Fowler–Zwaenepoel direct-dependency technique does not allow the transitive dependencies to be captured in real time during the execution of processes. In addition, a process must observe an event (i.e., update and record its dependency vector) after receiving a message but before sending out any message.
- Otherwise, during the reconstruction of a vector timestamp from the direct-dependency vectors, all the causal dependencies will not be captured.
- If events occur very frequently, this technique will require recording the history of a large number of events.

Jard–Jourdan's adaptive technique

- In the Jard–Jourdan's technique, events can be adaptively observed while maintaining the capability of retrieving all the causal dependencies of an observed event. (Observing an event means recording of the information about its dependencies.)
- This method uses the idea that when an observed event 'e' records its dependencies, then events that follow can determine their transitive dependencies, that is, the set of events that they indirectly depend on, by making use of the information recorded about e.

Jard–Jourdan's adaptive technique

- The reason is that when an event e is observed, the information about the send and receive of messages maintained by a process is recorded in that event and the information maintained by the process is then reset and updated.
- So, when the process propagates information after e, it propagates only history of activities that took place after e.
- The next observed event either in the same process or in a different one, would then have to look at the information recorded for e to know about the activities that happened before e.

Jard–Jourdan's adaptive technique

- This method still does not allow determining all the causal dependencies in real time, but avoids the problem of recording a large amount of history which is realized when using the direct dependency technique.

Jard–Jourdan's adaptive technique

- To implement the technique of recording the information in an observed event and resetting the information managed by a process, Jard–Jourdan defined a pseudo-direct relation « on the events of a distributed computation as follows:

Jard–Jourdan's adaptive technique

- If events ei and ej happen at process pi and pj , respectively, then ej « ei iff there exists a path of message transfers that starts after ej on the process pj and ends before ei on the process ei such that there is no observed event on the path.
- The relation is termed pseudo-direct because event ei may depend upon many unobserved events on the path, say ue1, ue2, ……./ , uen, etc., which are in turn dependent on each other.

Jard–Jourdan's adaptive technique

- If ei happens after uen, then ei is still considered to be directly dependent upon ue1, ue2,…… , uen, since these events are unobserved, which is a falsely assumed to have direct dependency.
- If another event ek happens after ei, then the transitive dependencies of ek on ue1, ue2,…… , uen can be determined by using the information recorded at ei and ei can do the same with ej .

Jard–Jourdan's adaptive technique

The technique is implemented using the following mechanism

- The partial vector clock p_vt_i at process pi is a list of tuples of the form (j, v) indicating that the current state of pi is pseudo-dependent on the event on process pj whose sequence number is v.

Initially, at a process p_i: $p\_vt_i = \{(i, 0)\}$.

Let $p\_vt_i = \{(i_1, v_1), \ldots, (i, v), \ldots (i_n, v_n)\}$ denote the current partial vector clock at process P_i

Let e_vt_i be a variable that holds the timestamp of the observed event.

eX_pt_n denotes the timestamp of the Xth observed event at process p_n.

Jard–Jourdan's adaptive technique

(i) Whenever an event is observed at process p_i, the contents of the partial vector clock p_vt_i are transferred to e_vt_i and p_vt_i is reset and updated as follows:

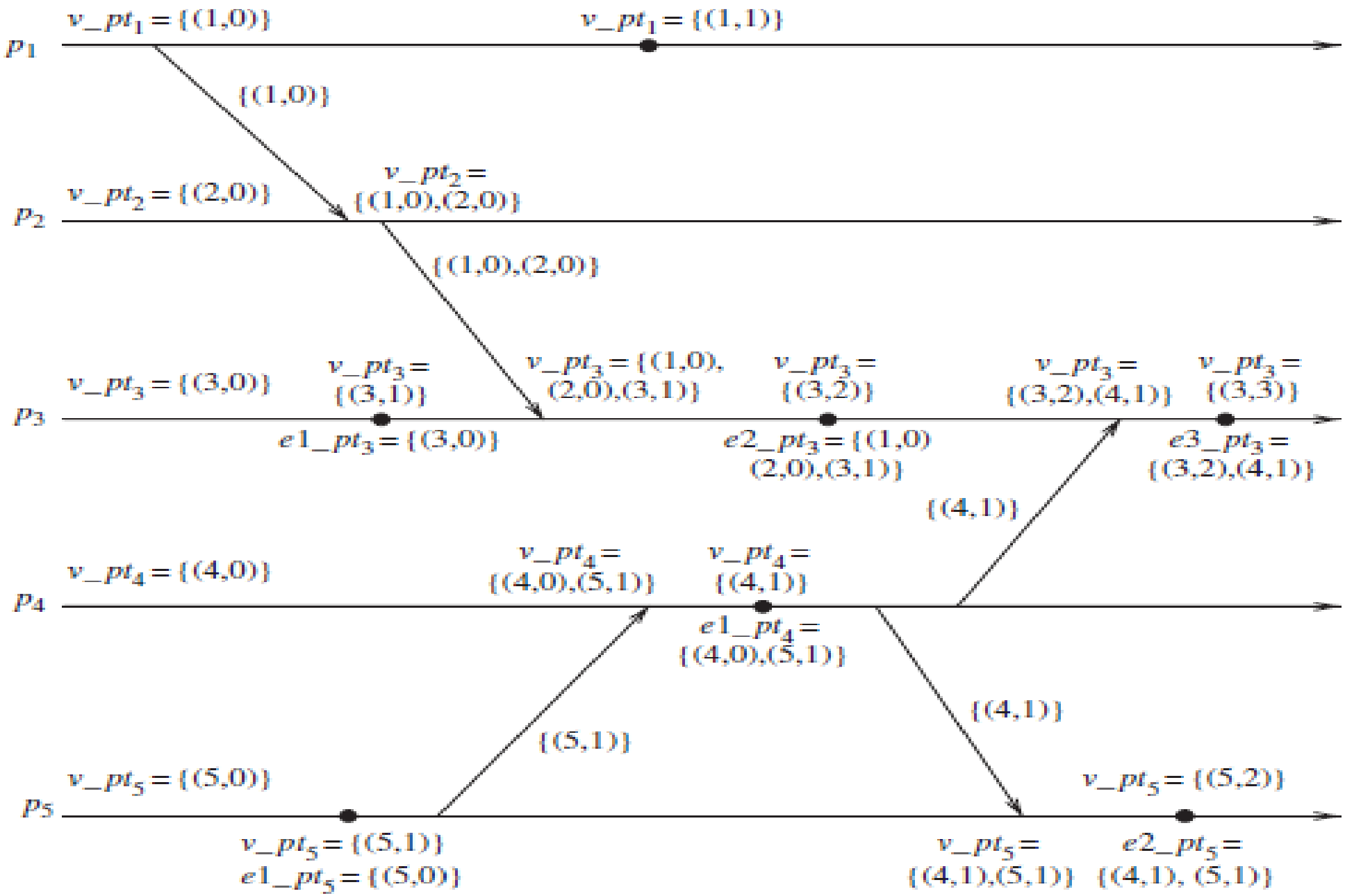
$$e\_vt_i = \{(i_1, v_1), \ldots, (i, v), \ldots, (i_n, v_n)\}$$
$$p\_vt_i = \{(i, v+1)\}.$$

(ii) When process p_j sends a message to p_i, it piggybacks the current value of p_vt_j in the message.

Jard–Jourdan's adaptive technique

(iii) When p_i receives a message piggybacked with timestamp p_vt, p_i updates p_vt_i such that it is the union of the following (let $p\_vt=\{(i_{m1}, v_{m1}),\ldots,(i_{mk}, v_{mk})\}$ and $p\_vt_i = \{(i_1, v_1), , \ldots, (i_l, v_l)\}$):

- all (i_{mx}, v_{mx}) such that $(i_{mx}, .)$ does not appear in v_pt_i;
- all (i_x, v_x) such that $(i_x, .)$ does not appear in v_pt;
- all $(i_x, max(v_x, v_{mx}))$ for all $(v_x, .)$ that appear in v_pt and v_pt_i.


$v\_pt_1 = \{(1,0)\}$
$v\_pt_1 = \{(1,1)\}$
p_1
$\{(1,0)\}$
$v\_pt_2 = \{(2,0)\}$
$v\_pt_2 = \{(1,0),(2,0)\}$
p_2
$\{(1,0),(2,0)\}$
$v\_pt_3 = \{(3,0)\}$
$v\_pt_3 = \{(3,1)\}$
$v\_pt_3 = \{(1,0),(2,0),(3,1)\}$
$v\_pt_3 = \{(3,2)\}$
$v\_pt_3 = \{(3,2),(4,1)\}$
$v\_pt_3 = \{(3,3)\}$
p_3
$e1\_pt_3 = \{(3,0)\}$
$e2\_pt_3 = \{(1,0)(2,0),(3,1)\}$
$e3\_pt_3 = \{(3,2),(4,1)\}$
$\{(4,1)\}$
$v\_pt_4 = \{(4,0)\}$
$v\_pt_4 = \{(4,0),(5,1)\}$
$v\_pt_4 = \{(4,1)\}$
p_4
$e1\_pt_4 = \{(4,0),(5,1)\}$
$\{(4,1)\}$
$\{(5,1)\}$
$v\_pt_5 = \{(5,0)\}$
$v\_pt_5 = \{(5,2)\}$
p_5
$v\_pt_5 = \{(5,1)\}$
$e1\_pt_5 = \{(5,0)\}$
$v\_pt_5 = \{(4,1),(5,1)\}$
$e2\_pt_5 = \{(4,1),(5,1)\}$

THANK YOU